CLASSIC Description and Reference Manual For the COMMON LISP Implementation Version 2.3

Lori Alperin Resnick
Alex Borgida
Ronald J. Brachman
Charles L. Isbell
Deborah L. McGuinness
Peter F. Patel-Schneider
Kevin C. Zalondek

December 31, 1995

Contents

1 Introduction

The CLASSIC Knowledge Representation System belongs to a new generation of description-based knowledge representation systems, emphasizing simplicity of the description language and completeness and tractability of its inference algorithms. The CLASSIC system goes significantly beyond existing systems like KANDOR in many important respects, including its language, integration with the host system, and treatment of individuals.

The first half of this manual is a description of the CLASSIC language: Section 2 describes the CLASSIC types: told and derived concepts and individuals, and roles; Section 2.6 discusses *rules*—the mechanism in CLASSIC for simple forward-chaining deductions; Section 3 informally discusses normalization, subsumption, and classification in CLASSIC; Section 4 discusses incompleteness in CLASSIC; Section 5 discusses updates in CLASSIC, including both addition and retraction of information for individuals; Section 6 describes error handling in CLASSIC; and Section 7 describes CLASSIC's explanation module. The appendices give a detailed specification of how to use the CLASSIC system: Appendix A shows the CLASSIC grammar; Appendix B discusses the CLASSIC reader; Appendix C defines some terms you will need to know in order to understand the function descriptions; Appendix D gives a description of the user functions; Appendix E describes the error codes produced by CLASSIC; Appendix F lists all the built-in concepts in CLASSIC; Appendix G describes the UNIX-style interface to CLASSIC; Appendix H describes the Description-Logic Knowledge Representation System Specification module in CLASSIC; Appendix I tells how to install and load CLASSIC; and Appendix J discusses the major recent additions/changes to CLASSIC.

2 Description of CLASSIC Types

In a frame system such as CLASSIC, a description of the world consists of a description of the *individuals* that exist in the world (the classes they belong to, and their properties), and the interrelationships between the individuals. Some individuals belong to the domain being modeled, i.e., particular wines, in a knowledge base that reasons about wines. (We refer to these individuals as being in the **CLASSIC**[1] realm, and we call them **CLASSIC** individuals.) Other individuals are internal individuals, used to help us describe **CLASSIC** individuals, i.e., ages may be represented as INTEGERS, and names may be represented as STRINGS. (We refer to these individuals as being in the **HOST** realm, and we call them **HOST** individuals.) In CLASSIC, *concepts* are descriptions that are applied to individuals (in the manner that one-place predicates are applied to constants in predicate logic). Concepts can be in either the **CLASSIC** realm or the **HOST** realm, depending on which type of individuals they describe. The *roles* in CLASSIC are separate entities that represent the properties of **CLASSIC** individuals. They are used to map **CLASSIC** individuals to other (**CLASSIC** or **HOST**) individuals. The roles of a **CLASSIC** individual can either be "filled" by individuals (called the role fillers) or have their potential fillers restricted by certain concepts (i.e., as type restrictions), or both, where the fillers and restrictions can

[1]**CLASSIC** refers to the realm, while CLASSIC refers to the knowledge representation system.

be in either the **CLASSIC** realm or the **HOST** realm. *Attributes* are special types of roles that have an implicit maximum number of fillers of `1`. See Appendix A for the grammar of all possible CLASSIC concept and individual descriptions.

2.1 Concepts

Concepts in CLASSIC are defined using compound expressions from a simple grammar (see Appendix A). We call these expressions *descriptions*, CLASSIC *descriptions*, or *concept descriptions*.

Concepts and descriptions in CLASSIC are divided into two realms: **CLASSIC** and **HOST**. **CLASSIC** concepts are used to represent classes of real-world objects of a domain, while **HOST** concepts are used to describe objects in the host language (currently LISP), such as NUMBERS, LISTS, ATOMS, STRUCTURES, and STRINGS.

The rest of this section describes the different operators used to form CLASSIC descriptions[2], some of which can be used to define both **CLASSIC** and **HOST** concepts (**AND** and **ONE-OF**), some that can be used to define only **CLASSIC** concepts (**TEST-C**, **ALL**, **AT-LEAST**, **AT-MOST**, **FILLS**, and **SAME-AS**), and some that can be used to define only **HOST** concepts (**TEST-H**, **MIN**, and **MAX**).

Note: any description that can legally be used to define a **CLASSIC** concept can also be used to describe a **CLASSIC** individual.

2.1.1 Named Concepts

The simplest type of CLASSIC description is a single symbol (designating a named concept). When a concept is defined, a name, such as `C`, is associated with its defining description. The user may define a new concept that incorporates `C`'s definition by applying the operators described below to `C`. For example, suppose that `VEGETARIAN`[3] has been defined as

[2]The term CLASSIC *description* or *concept description* refers to any subexpression used in defining a concept or an individual, while the term *concept* refers to an actual concept structure that has been created, and to which the user functions described in this manual can be applied. For example, suppose the concept `HAPPY-PARENT` is defined as a person with at least one child, and all of whose children are doctors:

```
(AND (AT-LEAST 1 child)
     (ALL child DOCTOR))
```

We would refer to `HAPPY-PARENT` as a concept, and any portion of its defining description (i.e., (**AT-LEAST** `1 child`)) as a CLASSIC *description*. In this section, there are a number of examples of CLASSIC descriptions. Note that you cannot just type in a concept description to LISP, but must call one of the functions **cl-define-concept**, **cl-define-primitive-concept**, or **cl-define-disjoint-primitive-concept** (see Section 2.1.7) to define a concept with the given description. However, for purposes of simplification, we often just present the concept descriptions, rather than showing a call to one of these functions each time.

[3]In this manual we use the following orthographic conventions:
`CONCEPT-NAME`: typewriter font, uppercase;
`Individual-Name`: typewriter font, lowercase with first letter uppercase;
`role-name`: typewriter font,lowercase;
function-name: small boldface font, lowercase;
CLASSIC-OPERATOR: small boldface font, uppercase.

"something all of whose food is of type PLANT". Then we may define VEGETARIAN-PERSON as "a VEGETARIAN and a PERSON", which is equivalent to "a PERSON, all of whose food is of type PLANT".

2.1.2 Conjunction (AND)

The **AND** operator forms the conjunction of some number of descriptions. The syntax is

(**AND** <descr>$^+$),[4]

where descr is a description—either the name of an existing concept, or a description formed using one of the operators described below. For example, a VEGETARIAN-PERSON might be someone who is both a VEGETARIAN and a PERSON:

(**AND** VEGETARIAN PERSON).

No components of an **AND** description may come from distinct members of disjoint primitive concepts of the same disjoint grouping (see Section 2.1.6), and still describe individuals. Therefore, all components of a *coherent* **AND** concept (i.e., an **AND** concept that may have instances) must come from the same realm, either **CLASSIC** or **HOST** (e.g., the concept described by (**AND** PERSON INTEGER) can have no instances), because CLASSIC-THING and HOST-THING are in the same disjoint grouping under THING (see Section 2.1.5). See also the discussions on *incoherent concepts*, which are not allowed as top-level concepts (Section 2.1.7), but are allowed within **ALL** restrictions (Section 2.1.3.1).

2.1.3 Role Restrictions

The five operators **ALL**, **AT-LEAST**, **AT-MOST**, **FILLS**, and **SAME-AS** form special types of concept descriptions known as *role restrictions*, and can only be used to describe **CLASSIC** individuals. Depending on the kind of role restriction, either the type of the fillers can be restricted (**ALL** restrictions), the number of fillers can be restricted (**AT-LEAST** and **AT-MOST** restrictions), some of the actual fillers can be specified (**FILLS** restrictions), or the filler of one attribute-path (list of attribute names—see Section 2.1.3.4) must be the same as the filler of another attribute-path (**SAME-AS** restrictions).

Important: Note that **HOST** concepts are prohibited from having role restrictions as components, just as **HOST** individuals may not have roles on them (i.e., you are not allowed to put a role on the integer 3).

2.1.3.1 Value Restriction (ALL)

A universal value restriction, or **ALL** restriction, has the syntax

(**ALL** <role-name> <descr>)

where a role with the given name must already exist.[5] An **ALL** restriction specifies that all the fillers of a particular role must be individuals described by a particular description. For an individual I to satisfy the value restriction (**ALL** r C_1), either r must be *closed* on I (see Section 2.2.3), and each of the fillers of r on I must be described by the concept C_1;

[4] Note: when we use a + after a type specifier in a definition, there must be at least one expression of that type, i.e., (**AND** <descr>$^+$) means that after the **AND** operator there must be at least one description. On the other hand, * means there may be zero or more expressions of that type.

[5] Note: all the examples given in the manual assume that any roles or concepts used in CLASSIC descriptions have been previously defined.

or there must be a derivable **ALL** restriction[6] on I, (**ALL** r C_2), such that C_1 *subsumes* (is more general than) C_2 (see Section 3.1.2).

For example, suppose that the VEGETARIAN concept has a restriction on the food role, which is the concept PLANT:

(**ALL** food PLANT).

This says that any individual that is a VEGETARIAN can have its food role filled only by individuals that are described by the PLANT concept, e.g., Leaf-Lettuce.

It is possible for the description in an **ALL** restriction to be *incoherent* (i.e., a description that can have no instances because it contains conflicting information). This means that the role specified in the restriction can have no fillers, which is fine as long as there is not a (positive) **AT-LEAST** restriction on the role (see Section 2.1.3.2). For example, suppose that MALE and FEMALE are two disjoint primitive concepts in the same disjoint grouping (i.e., no individual can be an instance of both concepts—see Section 2.1.6). Now suppose the concept ILLEGAL-CHILDREN were defined as

(**ALL** child (**AND** MALE FEMALE)).

Since no individual can be described by both MALE and FEMALE, this would be equivalent to the concept of someone with no children:

(**AT-MOST** 0 child)

(see Section 2.1.3.2).

2.1.3.2 Number (AT-LEAST, AT-MOST) Restrictions

An **AT-LEAST** restriction specifies the minimum number of fillers allowed for a given role on a concept or individual. The syntax is

(**AT-LEAST** <positive-integer> <role-name>)

For example, a PARENT might be defined to have at least 1 child:

(**AT-LEAST** 1 child).

For an individual I to satisfy the **AT-LEAST** restriction (**AT-LEAST** n_1 r), either r must have at least n_1 fillers on I, or there must be a derivable **AT-LEAST** restriction on I, (**AT-LEAST** n_2 r), such that n_2 is greater than or equal to n_1.

An **AT-MOST** restriction specifies the maximum number of fillers allowed for a given role on a concept or individual. The syntax is

(**AT-MOST** <non-negative-integer> <role-name>)

For example, an ORPHAN might be defined to have no parents:

(**AT-MOST** 0 parent).

For an individual I to satisfy the **AT-MOST** restriction (**AT-MOST** n_1 r), either r must be *closed* on I (see Section 2.2.3) and have at most n_1 fillers, or there must be a derivable **AT-MOST** restriction on I, (**AT-MOST** n_2 r), such that n_2 is less than or equal to n_1.

2.1.3.3 FILLS

The **FILLS** operator specifies that for the individual or concept being described, a particular role is filled by the specified individuals. The syntax is

(**FILLS** <role-name> <individual-name>$^+$)

where an individual name is either a symbol (in which case it refers to the **CLASSIC**

[6] A *derivable* restriction is one that was either stated as part of the definition, or can be deduced from other information.

object with that name, e.g., (**FILLS** `brother Fred Sam`) says that the `brother` role is filled with the individuals `Fred` and `Sam`), or a string, a number, or a quoted Common LISP expression (in which case it refers to the **HOST** individual represented by the Common LISP expression, e.g., (**FILLS** `address "123 Main Street"`) says that the `address` role is filled with the given string). When a **FILLS** description is part of the definition of a concept, it means that for an individual to be an instance of the concept, it must have the given role filled with the given value(s) (and possibly other values also).

For example, we might define the concept `MALE-FRIEND-OF-SUE` as:

```
(AND PERSON
     (FILLS friend Sue)
     (FILLS gender Male))
```

where the filler of the `friend` role is the **CLASSIC** individual `Sue`, and the filler of the `gender` role is the **CLASSIC** individual `Male`. We could also define the concept `PERSON-NAMED-SAM` as:

```
(AND PERSON
     (FILLS first-name "Sam"))
```

where the filler of the `first-name` role is the Common LISP string `"Sam"`.

2.1.3.4 Equality Restrictions (SAME-AS)

A **SAME-AS** restriction (or equality restriction, or co-reference constraint) has the syntax

(**SAME-AS** <attribute-path> <attribute-path>)

where an *attribute-path* is just a non-empty list of attribute names. A **SAME-AS** restriction requires that the filler found by starting at the individual and following the first attribute-path is the same as the filler found by following the second attribute-path,[7] and that both paths have a filler. (Note: when a **SAME-AS** restriction is asserted on an object, all the attributes in both paths of the **SAME-AS** restriction get **AT-LEAST** restrictions of `1`.)

For an individual `I` to satisfy the **SAME-AS** restriction, (**SAME-AS** $\texttt{attribute-path}_1$ $\texttt{attribute-path}_2$), either the individuals must all exist when following the two attribute-paths and the final filler on each attribute-path must be the same individual, or there must be a derivable **SAME-AS** restriction or restrictions on the individual that imply this **SAME-AS** restriction.

For example, suppose the concept `CHILD-OF-MARRIED-PARENTS` is defined as

(**SAME-AS** `(mother husband) (father)`).

In order for an individual `Mary` to belong under the concept `CHILD-OF-MARRIED-PARENTS`, `Mary`'s `mother`'s `husband` must be her `father`.

Note: **SAME-AS** restrictions only take attribute-paths (i.e., paths of roles with at-most one filler), not arbitrary role-paths.

[7] Note: follow an attribute-path ($\texttt{a}_1$ ... $\texttt{a}_n$) from an individual $\texttt{I}_1$ as follows: For i=1..n, let $\texttt{I}_{i+1}$ be the filler of $\texttt{a}_i$ for $\texttt{I}_i$. The result of following the attribute-path is $\texttt{I}_{n+1}$.

2.1.4 Other Restrictions

2.1.4.1 Procedural tests

There are two types of descriptions that allow procedures to be used in specifying concepts: one used for describing **CLASSIC** individuals (**TEST-C**), and one used for describing **HOST** individuals (**TEST-H**). A test restriction requires that an individual must pass the test to satisfy the restriction. The syntax is

(**TEST-C** <fn> <arg>*) or (**TEST-H** <fn> <arg>*)

<fn> is a function in the **HOST** language (Common LISP) that will be run on a **CLASSIC** or **HOST** individual (for **TEST-C** or **TEST-H**, respectively).[8] The first argument that is passed to the function is always the individual being tested. The remaining arguments to the function will be the <arg>s, if any, and the function should expect the correct number of parameters for the arguments it will be passed.

The function, when applied to an individual, should be designed to return one of three possible values:

`NIL`: the individual is inconsistent with this restriction;

`?`: the individual is currently consistent with this restriction, but if information is added to the individual, the individual may become either inconsistent with the restriction or definitely described by the restriction;

`T` (or any non-`NIL` value other than `?`): the individual definitely satisfies this restriction.

When the system is told that an individual `I` is described by a test description, then the individual is consistent as long as the test function returns a non-`NIL` value. However, when an individual is being classified, it will not be classified under a test description if the test function returns `NIL` or `?`.

To guarantee correct behavior, the user must follow these requirements (they are not checked for): (Note: for purposes of this discussion, `T` refers to anything but the symbols `NIL` or `?`.)

- When information is added to an individual `I`, the result of applying a test function `f` to `I` may change from `?` to either `T` or `NIL`, but it may *never* change from `T` to `NIL`. (This requirement does not hold during the *retraction* of information from the individual—see Section 5.2.)
- When a **TEST-H** concept is applied to a **HOST** individual, the function must return either `NIL` or `T` (or any non-`?` value), because **HOST** individuals are unchanging, by definition.

Normally, test descriptions are not used alone, but are used as part of **AND** descriptions (see Section 2.1.2). For example, suppose the user defines the concept `EVEN-INTEGER` as a subconcept of `INTEGER` (a built-in concept—see Section 2.1.5 below), with a predicate (**evenp**) that decides whether or not the integer is even:

(**AND** `INTEGER` (**TEST-H** **evenp**)).

[8] <fn> must be a Common LISP object that can be passed to **funcall**, i.e., it must be a symbol bound to a function, a function, or a lambda expression.

The system will first test to see if the individual is an integer, and then an even integer. Test functions can depend on this and can safely perform operations that are dangerous to objects that do not belong to the parents (i.e., the function **evenp** would blow up if called on a `STRING`, but this will never happen since before running the **evenp** function on an individual, CLASSIC will first run the **integerp** function, which the concept `EVEN-INTEGER` inherits by being a subconcept of `INTEGER`, and if this test fails, **evenp** will not be called). Thus, the test restrictions inherited from the parent concepts are always guaranteed to be run before the told test restrictions on a concept, although there is no way of determining the ordering of the test restrictions between the different parents, or the ordering between the different told restrictions for a concept.

Suppose we wish to define an `EMPLOYEE` as a `PERSON` with at least `1 employer`, and who has an `age` between `18` and `65`. First we write a LISP function, **employee-age-test**, which returns `T` if its argument is between `18` and `65`, and `NIL` otherwise.

```
(defun employee-age-test (age)
  (<= 18 age 65))
```

Next we define the concept `EMPLOYEE` by the following concept description:

```
(AND PERSON
     (AT-LEAST 1 employer)
     (ALL age (AND INTEGER
                   (TEST-H employee-age-test))))
```

Note that we must specify `INTEGER` as part of the **ALL** restriction, because the <= function would blow up if it were applied to anything that is not an `INTEGER`. Note: this concept could have been defined using an interval (see Section 2.1.4.2), but a test function was used for illustrative purposes.

The following is a more complicated example, requiring a three-valued test function. Suppose we wish to define a `SUCCESSFUL-PARTY` as a `PARTY` where the number of male guests is the same as the number of female guests. We first create a general function, **same-number-fillers**, which takes an individual and two role-names. It returns `T` if both roles provably have exactly the same number of fillers, `NIL` if both roles provably have a different number of fillers, and `?` otherwise. Note that "provably" here involves the roles being closed, or some provable generic relation between the **AT-LEAST** and **AT-MOST** restrictions on the roles.

```
(defun same-number-fillers (ind role-name-1 role-name-2)
 (let*
  ((role-1 (cl-named-role role-name-1))            ; the roles with the given names
   (role-2 (cl-named-role role-name-2))
   (min-1 (cl-at-least ind role-1))                ; the AT-LEAST restrictions
   (min-2 (cl-at-least ind role-2))
   (max-1 (cl-at-most ind role-1))                 ; the AT-MOST restrictions
   (max-2 (cl-at-most ind role-2))
   (number-fillers-1 (length (cl-fillers ind role-1))) ; the current number of fillers
   (number-fillers-2 (length (cl-fillers ind role-2)))
   (closed?-1 (cl-ind-closed-role? ind role-1))    ; are the roles closed?
   (closed?-2 (cl-ind-closed-role? ind role-2)))
  (cond
   ((or (and closed?-1 closed?-2  ; both roles provably have the same number of fillers
             (= number-fillers-1 number-fillers-2))
        (= min-1 max-1 min-2 max-2))
    T)
   ((or (and closed?-1 closed?-2) ; both roles provably have a different number of fillers
        (> min-1 max-2)
        (> min-2 max-1))
    NIL)
   (t '?)))) ; unknown
```

Next we define the concept `SUCCESSFUL-PARTY` by the following description:

(**AND** `PARTY` (**TEST-C** **same-number-fillers** `male-guests female-guests`)).

If the individual `Party-1` is a `PARTY` with exactly `8 male-guests` and exactly `8 female-guests` (whether the guests are known, or just known to exist), then it will be classified as a `SUCCESSFUL-PARTY`. If the individual `Party-2` is asserted to be a `SUCCESSFUL-PARTY`, then as long as the **same-number-fillers** test does not return `NIL` on `Party-2`, it is consistent for it to be a `SUCCESSFUL-PARTY`.

Note: The system does not do any analysis of procedures to determine whether one test description is more specific than another.

Important note: there are certain requirements the user must adhere to when writing any procedural test functions. This includes functions for **TEST-C**/**TEST-H**, both in concept and individual definitions, in rule filters, and as consequents of simple rules (see Section 2.6). The safest thing for the user to do within a test function is to just ask questions about the properties of the current individual (see the **same-number-fillers** test function above). The user can assume that at the time of classification, the individual has been completely normalized (see Section 3), and all local inferences have been done (i.e., those not involving any other individuals, and those that don't depend on the firing of rules).

It is not as safe to access the properties of related individuals in the knowledge base, since not all inferences may have been done when the test function is run. In addition, if the properties of related individuals change, then unless a test function returns dependencies (see the discussion for advanced CLASSIC users at the end of this section), there is no way for CLASSIC to know that it must reclassify the affected individual (since a test function is like a black box, and CLASSIC cannot automatically calculate dependency relationships based on information in a test function).

For example, suppose we wish to define the `DOCTOR-CHILD` concept as someone who has at least 1 child who is a doctor. First we define a test function, **doctor-child-fn**, which checks to see if any filler of the `child` role is an instance of the `DOCTOR` concept.

```
(defun doctor-child-fn (ind)
  (let* ((child-role (cl-named-role 'child))
         (children (cl-fillers ind child-role))
         (doctor-concept (cl-named-concept 'DOCTOR))
         (doctor-child
          (member-if #'(lambda (child)
                          (cl-instance? child doctor-concept))
                     children)))
    (if doctor-child t '?)))
```

Then we define the concept `DOCTOR-CHILD` as (**TEST-C doctor-child-fn**). If we create `Mary` as a `DOCTOR`, and then create `John` as someone whose `child` is `Mary`, then `John` will correctly get classified under `DOCTOR-CHILD`. However, if we create `Fred` as someone whose `child` is `Harry`, and then later assert that `Harry` is a `DOCTOR`, then CLASSIC will not know to reclassify `Fred`, and thus will not discover that `Fred` is now a `DOCTOR-CHILD`. At some later point, if information is added to `Fred`, he will get reclassified under `DOCTOR-CHILD`.

It is acceptable to produce side effects within test functions, *as long as the side effects are outside of* CLASSIC. However, the user cannot count on when and how often a test function will get run. Thus, it is not meaningful to increment a counter every time a test function is run. *The user must not write test functions which produce side effects within* CLASSIC. It is not acceptable for a test function to call any CLASSIC addition/retraction functions (**cl-ind-add**, **cl-ind-remove**, **cl-add-rule**, etc.). This could cause all sorts of problems.

These same restrictions on side effects apply also to the functions for filler and description rules (see Section 2.6).

Dependencies for Test Functions
Warning: For Advanced CLASSIC Users Only

During classification, CLASSIC calculates and keeps track of certain dependency relationships. For example, if all of `Mary`'s `children`'s `children` are known, and they are all `ATHLETES`, then `Mary` would be classified under the concept `GRANDPARENT-OF-ATHLETES`. However, if one of her grandchildren, say `Fred`, stops being an `ATHLETE` at any point (the parent concept `ATHLETE` is removed from him), CLASSIC must know to reclassify `Mary` so that she is no longer under the concept `GRANDPARENT-OF-ATHLETES`. Thus, when `Mary` is classified under `GRANDPARENT-OF-ATHLETES`, all her `children` and grandchildren are stored as *negative dependencies* of `Mary`, which means that if any information is `removed` from them, `Mary` needs to be reclassified.

In addition, if Mary is not classified under GRANDPARENT-OF-DOCTORS, because two of her grandchildren, Lou and Sarah, are not known to be DOCTORS, then Lou and Sarah are stored as *positive dependencies* of Mary, which means that if any information is added to them, Mary needs to be reclassified.

Since test restrictions are basically black boxes to CLASSIC, i.e., CLASSIC has no way of knowing what goes on inside them, CLASSIC cannot automatically calculate dependencies when an individual is being tested for subsumption against a concept containing a test restriction. Thus, if a test restriction uses information about other individuals when it is run on an individual, then if things change about those other individuals, CLASSIC somehow needs to know to reclassify that individual.

This can be done if the user's test functions return dependency information. The first return value from a test function is always T, NIL, or ?, as stated before. However, a test function may also return two additional values. The second and third values are the positive and negative dependencies (pdeps and ndeps), respectively.

As an example of a test function which returns dependencies, see the function **cl-test-all-closed?**, in **library.lisp**, which takes a list of roles and/or role-paths, and returns whether or not all the role-paths are closed on the individual. If all role-paths are closed, then any intermediate individual along any path is on the ndeps list (if we remove information from one of these individuals, specifically, that the role is closed, then this test function should no longer return T). Otherwise, if any role-path is non-closed, then

- The individual which doesn't have the closed role is the only individual in the pdeps list.
- All intermediate individuals leading up to and including that individual are on the ndeps list (if any of these fillers is removed, then the pdeps need to be recalculated).

2.1.4.2 Intervals (MIN, MAX)

A **MIN** restriction specifies that an individual must be a number with a given lower bound. The syntax is

(**MIN** <number>)

For example, the description (**MIN** 5) specifies that the value must be greater than or equal to 5.

A **MAX** restriction specifies that an individual must be a number with a given upper bound. The syntax is

(**MAX** <number>)

For example, the description (**MAX** 5) specifies that the value must be less than or equal to 5.

Intervals are usually used for **ALL** restrictions. For example, the following restricts someone's age to be an integer between 18 and 65:

(**ALL** age (**AND** INTEGER (**MIN** 18) (**MAX** 65))).

A **HOST** description can contain a **MIN** restriction, a **MAX** restriction, or both.

2.1.4.3 ONE-OF

A **ONE-OF** restriction enumerates a set of individuals, which are the only possible instances of the description. The syntax is

(**ONE-OF** <individual-name>*)
For example, we can state that the color of a wine can be only white, red, or rose: (**ONE-OF** White Red Rose).

The members of a **ONE-OF** description must all be from the same realm, **CLASSIC** or **HOST**, otherwise an error is signaled. A **ONE-OF** restriction with no individuals, (**ONE-OF**), is treated as an incoherent concept (a concept which can have no instances), and is illegal at the top level. (It can be used within an **ALL** restriction—see Section 2.1.3.1.)

A **ONE-OF** concept E_1 will be classified below another **ONE-OF** concept E_2 if E_1 is a subset of E_2. For example,

(**ONE-OF** Red Blue)

will be classified below

(**ONE-OF** Red Blue Green),

since the first concept is more specific than the second.

An individual that is a member of the set described by a **ONE-OF** concept will always satisfy that **ONE-OF** concept. For example,
the individual "who" will satisfy the concept

(**ONE-OF** "who" "am" "I").

An **AND** concept E_3 (see Section 2.1.2) which is the combination of a **ONE-OF** concept and additional restrictions will be classified below the **ONE-OF** concept. For example,

(**AND** MAMMAL (**ONE-OF** Joe Mary Tweety))

will be classified below

(**ONE-OF** Joe Mary Tweety).

However, the individuals mentioned in the **ONE-OF** restriction on the **AND** concept might not all be instances of the **AND** concept. If Tweety is not known to be a MAMMAL, it will not get classified below (**AND** MAMMAL (**ONE-OF** Joe Mary Tweety)).

See Section 4 for a discussion of how CLASSIC is incomplete with respect to subsumption of **ONE-OF** restrictions.

2.1.5 Built-In Concepts

The CLASSIC system comes with a number of built-in concepts. The top concept in the concept hierarchy is THING. Underneath THING in the concept hierarchy are the two disjoint primitives CLASSIC-THING and HOST-THING (in the realm disjoint grouping).[9] No other concepts and no individuals may fall directly underneath THING. All **CLASSIC** individuals are instances of (are described by) CLASSIC-THING, and all **HOST** individuals are instances of HOST-THING. Since no concept or individual can be described by two disjoint primitives in the same disjoint grouping, no concept or individual can fall under both CLASSIC-THING and HOST-THING; i.e., no concept or individual can be in both the **CLASSIC** realm and the **HOST** realm.

There are certain built-in concepts under HOST-THING in the concept hierarchy. In the Common LISP implementation, where Common LISP is the host language, the **HOST** concepts NUMBER, LIST, ATOM, STRUCTURE, and STRING are used to represent the corresponding built-in LISP object types NUMBER, LIST, ATOM, STRUCTURE, and STRING (This is just a sample of the LISP types actually represented.) Note the distinction between

[9]Note: HOST-THING is the only disjoint primitive concept in the **HOST** realm.

the LISP object types and the CLASSIC concepts used to represent them. The individuals `'foo` and `"Harry"` would be automatically classified under the concepts `ATOM` and `STRING` respectively. See Appendix F for a complete listing of built-in concepts.

There are built-in tests to determine whether or not a particular individual, such as `"Harry"`, is described by a concept such as `STRING`. These tests are based on LISP functions such as `stringp`, `atom`, `listp`, and `numberp`.

There is a built-in concept under `CLASSIC-THING`, `CONCEPT`, which is the parent of all meta individuals (see Section 2.4, and functions **cl-concept-meta-ind** and **cl-concept-meta-ind-concept** in Appendix D.6.2.3).

2.1.6 Defined, Primitive, Disjoint Primitive, and Temporary Concepts

There are three different types of concepts: defined concepts, primitive concepts, and disjoint primitive concepts.

A defined concept is one that is fully specified by its definition. Suppose that a `VEGETARIAN` is defined as someone who eats only plants, and a `GREEN-PLANT-VEGETARIAN` is defined as someone who eats only green plants. CLASSIC will infer that `GREEN-PLANT-VEGETARIAN` is a more specific concept than (is subsumed by) `VEGETARIAN` (see Section 3). Also, if an individual `Fred` is known to eat only `Squash`, CLASSIC will determine that `Fred` must be a `VEGETARIAN`.

Primitive concepts, on the other hand, cannot be fully specified by necessary and sufficient conditions. A concept C_1 will not be classified below a concept, C_2, that contains a primitive concept P_2, unless C_1 is also known to contain P_2 (see section 3). In other words, it must be explicitly indicated within any concept which primitives it belongs below. No deductions, other than inheritance, can determine that a concept `C` is subsumed by a primitive concept `P`, when `P` is not part of the definition of `C`.

For example, suppose the concepts `MAMMAL` and `INVERTEBRATE` are defined as primitive concepts under `ANIMAL`, and the concept `DOG` is defined as a primitive concept under `MAMMAL`. We can deduce that `DOG` is subsumed by both `MAMMAL` (this was stated) and `ANIMAL` (this was inherited), but there is no way we could deduce that `DOG` is subsumed by `INVERTEBRATE`.

Here are two examples of primitive concepts with their defining descriptions:

```
MAMMAL: ANIMAL
WINE: (AND DRINK
            (AT-LEAST 1 color) (AT-MOST 1 color)
            (ALL color (ONE-OF Red White Rose))
            (AT-LEAST 1 grape) (ALL grape GRAPE)
            (AT-LEAST 1 sugar) (AT-MOST 1 sugar)
            (ALL sugar (ONE-OF Dry Off-Dry Sweet)))
```

A disjoint primitive concept is just like a primitive concept, except that it belongs to one or more disjoint groupings, and any concepts within the same disjoint grouping are known to be disjoint from each other. Thus, no individual can be described by two disjoint primitive concepts in the same disjoint grouping.

The following example is depicted as a hierarchy in Figure 1, where the concepts within a given arc represent members of a disjoint grouping: Suppose the concepts `MAN` and `WOMAN`

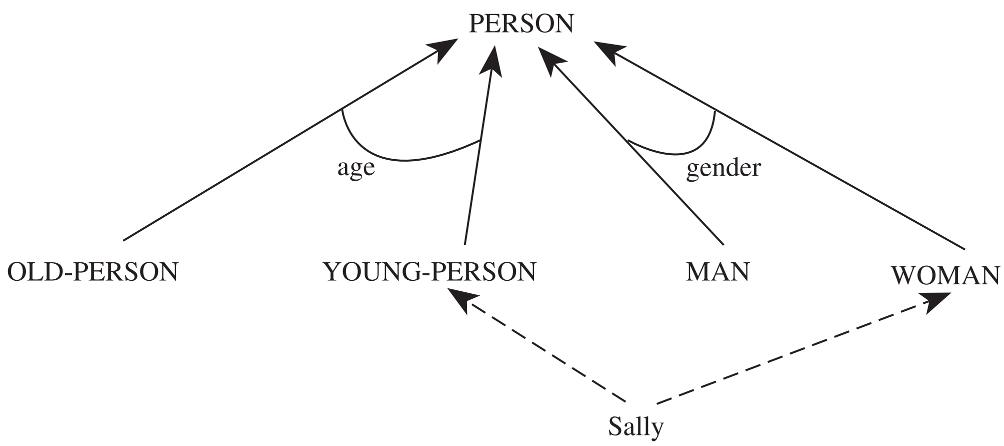

Figure 1: A hierarchy with disjoint groupings.

are defined as disjoint primitive concepts under `PERSON`, in the `gender` disjoint grouping, and the concepts `OLD-PERSON` and `YOUNG-PERSON` are defined as disjoint primitive concepts under `PERSON`, in the `age` disjoint grouping. This says that no individual can be both a `MAN` and a `WOMAN`, and no individual can be both an `OLD-PERSON` and a `YOUNG-PERSON`. However, there could be an individual that is both a `WOMAN` and a `YOUNG-PERSON`. Note that no exhaustiveness assumption is made when reasoning with disjoint primitive concepts (i.e., there could be individuals that are described by `PERSON`, but which are not described by either `OLD-PERSON` or `YOUNG-PERSON`). There is one exception: the built-in disjoint primitive concepts `CLASSIC-THING` and `HOST-THING` are exhaustive underneath `THING` (see Section 2.1.5).

Note that primitive and disjoint primitive concepts can only be defined in the **CLASSIC** realm. The only **HOST** disjoint primitive concept is the built-in concept `HOST-THING`.

In addition to the named concepts described above, there are also *temporary* concepts. Temporary concepts are identical to the regular, defined concepts described above, except that they do not have names, they are not classified or stored in the concept hierarchy, and they cannot be the antecedents of rules (see Section 2.6). All the retrieval functions work on temporary concepts (except those that retrieve the rule consequents).

2.1.7 Defining Concepts

There are three functions for defining named concepts: **cl-define-concept**, **cl-define-primitive-concept**, and **cl-define-disjoint-primitive-concept**.

cl-define-concept is used to define a regular, *defined* concept. The syntax is

(**cl-define-concept** <symbol> <descr>)

where the symbol is the name of the concept being defined and the description is the concept definition, and is formed using the grammar in Appendix A. This function creates and classifies the concept, unless the new concept is found to be identical to an existing classified concept, in which case the symbol is added as a new name for the existing concept; or the concept is found to be *incoherent* (i.e., it can have no instances because it contains conflicting information), in which case the concept and its name are not added.

The following is an example of defining a `VEGETARIAN` as a `PERSON` who eats only `PLANT`s:

```
(cl-define-concept 'VEGETARIAN
                   '(AND PERSON (ALL food PLANT)))
```

The function **cl-define-primitive-concept** is used to define a primitive concept. The syntax is

(**cl-define-primitive-concept** <symbol> <**CLASSIC**-descr>)

where the symbol is the name of the concept being defined and the description is the concept definition, and is formed using the grammar in Appendix A.

`MAMMAL` is defined as a primitive concept under `ANIMAL` as follows:

(**cl-define-primitive-concept** `'MAMMAL 'ANIMAL`).

The function **cl-define-disjoint-primitive-concept** is used to define a disjoint primitive concept. The syntax is

(**cl-define-disjoint-primitive-concept** <symbol> <**CLASSIC**-descr>
<grouping symbol or list of grouping symbols>)

where the symbol is the name of the concept being defined, the description is the concept definition, and is formed using the grammar in Appendix A, and the disjoint grouping symbol(s) specify which disjoint grouping(s) the concept belongs to.

The following example defines the disjoint primitive concepts `MAN` and `WOMAN` under `PERSON` in the `gender` disjoint grouping, and `OLD-PERSON` and `YOUNG-PERSON` under `PERSON` in the `age` disjoint grouping:

```
(cl-define-disjoint-primitive-concept 'MAN 'PERSON 'gender)
(cl-define-disjoint-primitive-concept 'WOMAN 'PERSON 'gender)
(cl-define-disjoint-primitive-concept 'OLD-PERSON 'PERSON 'age)
(cl-define-disjoint-primitive-concept 'YOUNG-PERSON 'PERSON 'age)
```

Sometimes it might be appropriate to define temporary concepts: e.g., if we are trying out certain parameters for a concept's definition, and we are not certain that the concept is in its final form, or if we only need a concept for a short time, and we do not wish to clutter up the knowledge base. There are two functions for defining temporary concepts: **cl-parse-expression** and **cl-normalize-concept**. The syntax for **cl-parse-expression** is

(**cl-parse-expression** <descr>)

where the description is the concept definition. This returns a temporary (unclassified) told concept. The syntax for **cl-normalize-concept** is

(**cl-normalize-concept** <unclassified told concept>)

where the unclassified told concept could be one that was returned by **cl-parse-expression**. It returns a temporary (unclassified) derived concept. (Note that it is possible for **cl-normalize-concept** to return a classified concept—see Section 2.1.8.) See Section 2.3 for a description of told and derived concepts.

2.1.8 Classified and Unclassified Concepts

A classified concept is one whose most specific subsumers (its parents) and most general subsumees (its children) have been calculated and stored in the concept hierarchy (see Appendix C). All named concepts (those defined by **cl-define-concept**, **cl-define-primitive-concept**, and **cl-define-disjoint-primitive-concept**—see Section 2.1.7) are classified. **ALL** restrictions (returned by **cl-all** and **cl-told-all**—see Appendices D.5.2 and D.5.3), and the consequents and filters of rules (see Section 2.6, and **cl-told-rule-consequent**, **cl-rule-consequent**, **cl-told-rule-filter**, and **cl-rule-filter** in Appendix D.9.2), are either classified or unclassified, depending on whether or not they are named concepts.

Temporary concepts, usually the concepts returned by **cl-parse-expression** and **cl-normalize-concept** (see Sections 2.1.6 and 2.1.7, and Appendix D.6.3.1), are unclassified concepts. If **cl-parse-expression** is called with the name of an existing concept as its defining description, it will return an unclassified told concept, but if **cl-normalize-concept** is then called on this unclassified told concept, it will return an existing classified derived concept, i.e.,

```
(setq c1 (cl-parse-expression 'CLASSIC-THING))
```

returns an unclassified told concept, while

```
(cl-normalize-concept c1)
```

returns the built-in derived concept `CLASSIC-THING`. Also, if **cl-normalize-concept** is called on a told classified concept, it will return the corresponding derived concept. (See Section 2.3 for a description of told and derived concepts.)

Since unclassified concepts are not stored in the concept hierarchy, they will not be returned by any retrieval functions which access information in the hierarchy (i.e., functions to return parents, ancestors, children, descendants); however, any concept retrieval function may be called on an unclassified concept (except as noted—see Appendix D.6.2). No rules may contain an unclassified concept as antecedent (see Section 2.6).

2.2 Individuals

Individuals are specific instances of CLASSIC concepts. Just as in the case of concepts, individuals are divided into two realms: **CLASSIC** and **HOST**. **CLASSIC** individuals are used to represent the real-world objects of a domain, while **HOST** individuals are objects in the host language (currently LISP). For example, the **CLASSIC** individual `Fred` might be an instance of the primitive **CLASSIC** concept `PERSON`, while the **HOST** individuals `10000` and `"Harry"` would be instances of the **HOST** concepts `NUMBER` and `STRING`, respectively, which were discussed in Section 2.1.5. **HOST** individuals cannot be created or modified, but they can be used in **FILLS** and **ONE-OF** descriptions.

The language of operators for **CLASSIC** individual descriptions is identical to that for **CLASSIC** concept descriptions, i.e., the user can use any of the concept operators described in Section 2.1 above (except **TEST-H**, **MIN**, and **MAX**) to describe a **CLASSIC** individual. In addition, by means of a function call, a role on an individual can be closed. This asserts that the currently known fillers of the role on the individual are provably all the fillers. This cannot be stated in an individual description. See Appendix A for all possible individual descriptions. This section discusses some of the differences in the effects of the operators on individuals, and gives some examples.

2.2.1 Equality Restrictions (SAME-AS)

During the normalization process (see Section 3.2.1), a **SAME-AS** restriction may cause an individual at the end of one of the attribute-paths to be filled with the individual at the end of the other attribute-path. For example, suppose that `John` is defined as a `PERSON` whose `mother`'s `age` is the same as his `father`'s `age`, and whose `mother` is `Mary`, who is known to have `age 35`, and whose `father` is `Bill`:

```
(AND PERSON
      (SAME-AS (mother age) (father age))
      (FILLS mother Mary)      ;; Mary is known to have age 35
      (FILLS father Bill))
```

Since the first path is completely known (his `mother` is `Mary`, and her `age` is `35`), and the second path is completely known except for the very last filler (his `father` is `Bill`, and `Bill`'s `age` is not known), we fill in the last role in the second path with the last role in the first path, i.e., `Bill`'s age is now known to be `35`.

2.2.2 Creating an individual

The function **cl-create-ind** is used to create a **CLASSIC** individual. The syntax is
(**cl-create-ind** <symbol> <**CLASSIC**-descr>)
where the symbol is the name of the individual being created, and the description is the description of the individual, and is formed using the grammar in Appendix A. For example, to create the individual `Mary` as a `PERSON`, we would say:
(**cl-create-ind** `'Mary 'PERSON)`.
To create the individual `Sam` as a `PERSON`, both of whose parents are `DOCTORS`, and who has a brother `Fred` and at least 1 sister, we would say:

```
(cl-create-ind 'Sam '(AND PERSON
                          (ALL parent DOCTOR)
                          (FILLS brother Fred)
                          (AT-LEAST 1 sister)))
```

2.2.3 Closing a role

There is one other thing a user may say about an individual—that a particular role is closed, i.e., it can have no more fillers. For reasons beyond the scope of this manual[10], there is no **CLOSE** operator as part of the language, but instead, there is a function, **cl-ind-close-role**, which is used for this purpose. Its syntax is
(**cl-ind-close-role** <individual> <role>)
For example, if `Sue` is `80` years old, has child `Joshua`, and has no other children, we would create `Sue` as follows:

```
(cl-create-ind 'Sue '(AND PERSON
                          (FILLS age 80)
                          (FILLS child Joshua)))
```

and then call (**cl-ind-close-role** `@Sue @child`) (@ accesses the object with the given name (i.e., `Sue`)—see Appendix B).

[10]For the interested reader, the reason is roughly that this would make concept descriptions autoepistemic (and thus nonmonotonic).

Another way a role can get closed on an individual is when the **AT-MOST** restriction is reached by the number of fillers for the role, i.e., the role can hold no more fillers. For example, if the `age` role is an attribute (with an implicit **AT-MOST** restriction of `1`—see Section 2.5), and `Mary`'s `age` role is filled with `25`, then the `age` role on `Mary` is closed, because it can have no more fillers.

2.2.4 Open World Assumption

An open-world assumption is used in CLASSIC. Unless a specific role within an individual is closed by the user, or can be deduced to be closed, it is assumed that there may be more values associated with that role for the individual. This affects subsumption between individuals and concepts, which is described in Section 3.2.2.

If `John` has `children` `Mary`, `Fred`, and `Sam`, all of whom are `LAWYERS`, there is no way of knowing that *all* of `John`'s `children` are `LAWYERS` unless either we can independently derive that `John` satisfies this restriction, or we know that `John` has no more `children`. For example, if there is a derivable restriction on `John` of *at most* `3 children`, then the system deduces that `John` has no more `children` (because he already has `3`), and that his `child` role is closed, and thus we satisfy the **ALL** restriction.

By the same principle, suppose that `John` has `friends Jack` and `Jill`. We have no way of knowing that `John` has *at most* `2 friends` unless we can independently derive that `John` satisfies this restriction (either because the `friend` role had been closed for `John`, or because there was such an **AT-MOST** restriction derivable on `John`).

2.3 Told and Derived Concepts and Individuals

Both concepts and individuals really contain two kinds of information: told information—the information the user told CLASSIC about the object—and derived information—information that CLASSIC derived about the object, from rule-firings and propagations (for individuals), and from inheritance, classification, and combining information in a number of ways (for both concepts and individuals).

The told information is stored in a told concept or told individual, and the derived information is stored in a derived concept or derived individual, in a canonical, or normalized form (see Section 3). The functions **cl-told-concept** and **cl-told-ind** will take a derived concept/individual and return the corresponding told concept/individual, while the functions **cl-derived-concept** and **cl-derived-ind** will take a told concept/individual and return the corresponding derived concept/individual. An unclassified derived concept has no told concept associated with it, and an unclassified told concept has no derived concept associated with it.

There are a number of retrieval functions which operate on told objects (both concepts and individuals), and a number which operate on derived objects. These can be found in Appendix D.5.

When a CLASSIC object is returned within LISP, it is printed as @`c`{name}, @`tc`{name}, @`i`{name}, or @`ti`{name} for a derived concept, a told concept, a derived **CLASSIC** individual, or a told **CLASSIC** individual, respectively. (Note: when a role is returned, it is printed as @`r`{name}. Roles are described below.) The @ macro can also be used to *retrieve* these objects (see Appendix B).

2.4 Meta Individuals

Every derived concept can have a meta individual associated with it. The meta individual is retrieved by calling the function **cl-concept-meta-ind**. This function returns the existing meta individual associated with the concept, or creates a new one and returns it.

The meta individual associated with a concept is just a derived individual, and all the individual functions will work on it. This individual is created under the built-in concept `CONCEPT`. The meta individual for a concept refers to the extension of the concept (what you currently know about the concept), and it is up to the user to manage this information. For example, the user might want to store roles like `comment`, `average-age`, or `number-of-instances` on a meta individual.

The function **cl-concept-meta-ind-concept** takes a derived individual and returns the derived concept for which this individual is the meta individual, or `NIL` if the individual is not a meta individual.

2.5 Roles

Roles are used to specify the properties of an individual. Roles on individuals and concepts can be filled with specific values (see Section 2.1.3.3) or they can be restricted in several ways (see Section 2.1.3); roles on individuals can also be closed (see Section 2.2.3).

A role is an independent entity, and may have certain types of information associated with it. A role may be part of a role hierarchy, where for example the `relative` role would be more general than the `sister` role. Thus, the `sister` role would have the `relative` role as its *parent* (and the `relative` role would have the `sister` role as its *child*).

When roles which are part of a role hierarchy are restricted on a concept or individual, CLASSIC may infer restrictions on the other roles which are part of the role hierarchy. These inferences are as follows:

- Filler information is "inherited"[11] up the role hierarchy, e.g., if `Mary`'s `sister` role is filled with `Jane`, then her `relative` role gets filled with `Jane`.

- **ALL** restrictions are inherited down the role hierarchy, e.g., if **ALL** `Mary`'s `relative`s are `DOCTOR`s, then **ALL** her `sister`s are `DOCTOR`s.

- **AT-LEAST** restrictions are "inherited" up the role hierarchy, e.g., if `Mary` has **AT-LEAST** `2 sister`s, then she has **AT-LEAST** `2 relative`s.

- **AT-MOST** restrictions are inherited down the role hierarchy, e.g., if `Mary` has **AT-MOST** `5 relative`s, then she has **AT-MOST** `5 sister`s.

- Related roles can combine in several ways. As an example, suppose `Mary` has **AT-LEAST** `2 sister`s, all of whom are `FEMALE`, and **AT-LEAST** `3 brother`s, all of whom are `MALE`, where `FEMALE` and `MALE` are disjoint. Then CLASSIC infers that she has **AT-LEAST** `5 relative`s. (Note: in this example, the `sister` and `brother` roles must have the `FEMALE` and `MALE` restrictions, respectively, in order for CLASSIC to

[11] Typically, one thinks of information as being inherited down a hierarchy, but we also use the word "inherit" when information moves up a hierarchy.

infer that the **AT-LEAST** restriction on `relative` is the sum of the **AT-LEAST** restrictions on `sister` and `brother`, because CLASSIC does not assume that subroles of a particular role are disjoint.)

A role can have an *inverse*. For example, an `EMPLOYEE` would be in a `DEPARTMENT`, while a `DEPARTMENT` would contain `EMPLOYEE`s. Thus, the `in-department` role would be the inverse of the `has-employee` role.

A role can be defined as an *attribute*, which is a special kind of role with an implicit **AT-MOST** restriction of `1`. An attribute cannot be part of a role hierarchy, e.g., it cannot be either the parent or the child of another role. Only roles that have been defined as attributes may be used in **SAME-AS** restrictions—the use of regular roles will cause an error.

All this information is specified at the time the role is defined. The syntax for defining a role is

(**cl-define-primitive-role** <symbol> &key[12] parent inverse (attribute `NIL`)
(inverse-attribute `NIL`)
inverse-parent) where the symbol is the name of the role.

The parent keyword specifies the name of the parent role, which must already exist, and which must not be an attribute. The inverse keyword specifies the name of the inverse role, which must not yet exist. The defaults are that the role has no parent or inverse. The attribute and inverse-attribute keywords, which both default to `NIL`, specify whether or not the role and its inverse are attributes. The inverse-parent keyword specifies the name of the inverse role's parent, which must already exist, and which must not be an attribute. Note that if the role is its own inverse, the inverse-attribute and inverse-parent keywords are ignored. Thus, to create a `spouse` role, which is its own inverse, we would say

(**cl-define-primitive-role** `'spouse` :inverse `'spouse`)

To create a `husband` role, whose inverse role is `wife`, where both have the parent role `spouse`, we would say

(**cl-define-primitive-role** `'husband` :parent `'spouse` :inverse `'wife`
:inverse-parent `'spouse`)

A role `r` can be thought of as a two-place predicate: $\mathtt{r(individual_1,\ individual_2)}$, where the role-predicate is TRUE if `r` on $\mathtt{individual_1}$ is known to have the value (filler) $\mathtt{individual_2}$. For example, `brother(Mary, John)` would be TRUE if the `brother` role on the individual `Mary` is known to have the value `John`. A role on an individual may have any number of values, possibly none.

[12]The &key keyword, used in function headers, says that the following arguments may optionally be specified by the user. If an argument is a two-element pair surrounded by parentheses (), the first element is the argument name or type, and the second element is its default value, used when no argument is given. If you wish to specify a keyword argument, you must give the keyword name (starting with `:`), followed by the value. For example, if you call (**cl-define-primitive-role** `'brother`), the :attribute keyword has the value `NIL`. However, if you call (**cl-define-primitive-role** `'age` :attribute `T`), the :attribute keyword has the value `T`. Keyword arguments may be specified in any order.

2.6 Rules

CLASSIC provides three different types of forward-chaining rules: simple rules, which we will refer to as *rules*; *description rules*; and *filler rules*. All three types of rules are named, which provides easier access to a specific rule and allows a rule to be more easily identified. Also, each type of rule may optionally have a comment associated with it, which can be used to describe the rule in English. A rule may fire on an individual when a new individual is created, information is added to an existing individual, or a new rule is created.

A (simple) rule consists of an antecedent, which must be a classified concept, and a consequent, which is also a concept, but is not necessarily classified. Suppose there is a rule with concept C_1 as the antecedent and concept C_2 as the consequent (i.e., C_1 is the left-hand side of the rule, and C_2 is the right-hand side of the rule). Then as soon as individual I_1 is known to belong under concept C_1, the rule is "fired", and I_1 is deduced to belong under C_2. A concept or individual does not need to be described by the consequent C_2 in order to be classified under (described by) the antecedent C_1. However, once the rule is fired, the individual may be further classified based on the new information provided by the rule.

For example, suppose there is a rule with the `PERSON` concept as the antecedent and the following description as the consequent:

```
(AND (AT-LEAST 1 social-security-number)
     (ALL social-security-number SSNUM))
```

If `Mary` is created as a `PERSON`, then this rule "fires", and she is known to have a `social-security-number` that is of type `SSNUM`. If there happened to be a concept `THING-WITH-SOCIAL-SECURITY-NUMBER`, defined as

```
(AT-LEAST 1 social-security-number)
```

then `Mary` would ultimately get classified under this concept.

As a second example, suppose that in order to determine that someone is a `VEGETARIAN`, it is enough to know that the only kind of `food` she eats is of type `PLANT`. Suppose also that there is a rule stating that anyone who is a `VEGETARIAN` is known to be a `HEALTHY-THING`. Thus, in order to classify someone as a `VEGETARIAN`, the system is only required to know the kind of `food` she eats, but once she is known to be a `VEGETARIAN`, the system can immediately "fire" the rule and infer that she is a `HEALTHY-THING`.

A rule may optionally contain a filter, which is a concept, not necessarily classified, that must be satisfied in order for the rule to fire. By using a filter, the user may avoid creating an unnecessary named concept whose only purpose is to be the antecedent of a rule.

Suppose we want to create the `COLLEGE-BOUND-HS-STUDENT` rule, which states that all high school students with grade point averages above 3.0 are college material. Suppose that the concepts `HIGH-SCHOOL-STUDENT` and `COLLEGE-MATERIAL` exist. We would create a rule with antecedent `HIGH-SCHOOL-STUDENT`, consequent `COLLEGE-MATERIAL`, and filter `(`**ALL** `gpa (`**MIN** `3.0))`. There is no need to create the concept `HIGH-SCHOOL-STUDENT-WITH-GPA-ABOVE-3.0` in order to create this rule.

The function **cl-add-rule** creates a simple rule. Its syntax is

(**cl-add-rule** <symbol> <classified derived **CLASSIC** concept>
<**CLASSIC**-descr> &key filter comment)

It takes a name (a symbol), an existing classified concept for the antecedent, and a single **CLASSIC** description for the consequent, and it can optionally be given a filter, which is also a **CLASSIC** description, and a comment, which is a string. To create the `COLLEGE-BOUND-HS-STUDENT` rule above, this function would be called as follows:

```
(cl-add-rule 'COLLEGE-BOUND-HS-STUDENT @HIGH-SCHOOL-STUDENT
             'COLLEGE-MATERIAL
             :filter '(ALL gpa (MIN 3.0))
             :comment "A high school student with a gpa of at least 3.0 is
                       likely to go on to college.")
```

(the @ accesses the object with the given name—see Appendix B).

A *description rule* is similar to the simple rule described above, except that instead of the rule being defined with a consequent that is specified at rule creation time, and then merged into an individual when the rule is fired, the rule is defined with a function and arguments, which, when the rule is fired, generate a concept description to be merged into an individual. Thus, the description that is merged into an individual is not specified in advance, but can be based on information that is known about the individual at the time the rule is fired.

A description rule is defined with the function **cl-add-description-rule**. The syntax is

(**cl-add-description-rule** <symbol> <classified derived **CLASSIC** concept>
<fn> &key args filter comment)

It takes a name (a symbol), an existing classified concept for the antecedent, a Common LISP function[13], and optionally, a list of arguments for the function, a filter (a **CLASSIC** description), and a comment (a string). The function should take as arguments an individual, followed by as many arguments as are specified in the list of arguments. When the rule is fired on an individual, the function gets called with the appropriate arguments, and it produces a **CLASSIC** description, which is then merged into the individual. The function must be written so that once it fires on an individual, it will always produce the same result, even if new information is added to the individual. A filter should be used to keep the rule from firing until all the necessary information is available.

Suppose that we have a primitive concept `C` with the following description

```
(AND (AT-LEAST 1 a)
     (ALL a INTEGER)
     (AT-LEAST 1 b)
     (ALL b INTEGER))
```

where `a` and `b` are attributes. We want to enforce the constraint that the filler of `b` can be no larger than the filler of `a`.

To implement this constraint, we first write the function **generate-b-max** to generate an **ALL** restriction on `b`:

[13]Like the **TEST-C/TEST-H** functions, it must be a symbol bound to a function, a function, or a lambda expression.

```
(defun generate-b-max (ind)
   (let* ((a-role (cl-named-role 'a))
          (a-filler (first (cl-fillers ind a-role))))
     `(ALL b (MAX ,a-filler))))
```

We must define a filter function which determines that there is currently a filler for a given role. This function returns `T` if there is currently a filler, `NIL` if there can be no fillers, and `?` otherwise.

```
(defun cl-test-role-filled? (ind role-name)
  (let* ((role (cl-named-role role-name)))
    (cond ((cl-fillers ind role) T)
          ((cl-ind-closed-role? ind role) NIL)
          (t '?))))
```

We then define the `generate-b-max-rule` description rule:

```
(cl-add-description-rule 'generate-b-max-rule @C{C}
   'generate-b-max :filter '(TEST-C cl-test-role-filled? a))
                   :comment "b's filler is <= a's filler."
```

Note: the user must be careful in producing the description, to generate unquoted *names* of all the CLASSIC objects referred to. Thus, if the arguments to the function are an individual and a role, say `@i{Mary}` and `@r{sibling}`, then the classic expression generated must refer to the symbols `Mary` and `sibling`, rather than the individual and the role, so they must call **cl-ind-name** and **cl-role-name** to get these symbols.

A *filler rule* is like a description rule, except that in addition to the function and arguments, it takes a role, and the function generates a list of fillers for the role when the rule is fired.

A filler rule is defined with the function **cl-add-filler-rule**. The syntax is

```
(cl-add-filler-rule <symbol> <classified derived CLASSIC concept>
                    <role> <fn> &key args filter comment)
```

It takes a name (a symbol), an existing classified concept for the antecedent, a role, a Common LISP function, and optionally, a list of arguments for the function, a filter (a **CLASSIC** description), and a comment (a string). The function should take as arguments an individual, a role, and as many arguments as are specified in the list of arguments. When the rule is fired on an individual, the function gets called with the appropriate arguments, and returns a list of derived individuals. These individuals then get added as fillers of the specified role on the individual. The function must be written so that once it fires on an individual, it will always produce the same list of individuals, even if new information is added to the individual.

For example, suppose we know that a certain type of rectangle, `RECT1`, has a length which is 4 times its width, where `length` and `width` are attributes, and `RECT1` is a concept. We would define the function **calculate-length** as follows:

```
(defun calculate-length (ind length-role)
  (declare (ignore length-role))
  (let* ((width-role (cl-named-role 'width))
         (width (first (cl-fillers ind width-role))))
    (list (* width 4))))
```

We would then add the rule as follows:

```
(cl-add-filler-rule 'rect1-length @c{RECT1} @r{length}
   'calculate-length :filter '(TEST-C cl-test-role-filled? width))
```

An illegal use of a filler rule would be a rule that fires, and the function generates NIL as the result, but when more information is added to the individual, the function generates a list of individuals. In this case, a filter should be used to keep the rule from firing until enough information is in the individual for the function to produce the final result.

We can also remove rules from the knowledge base. The function **cl-remove-rule** removes a single rule. The syntax is

(**cl-remove-rule** <rule>)

For example, we could remove the COLLEGE-BOUND-STUDENT rule as follows:

```
(cl-remove-rule (cl-named-rule 'COLLEGE-BOUND-STUDENT))
```

(the function **cl-named-rule** accesses the rule with the given name—see Appendix D.9.2.1).

The function **cl-remove-all-rules** takes a classified derived **CLASSIC** concept, and removes all the rules with this concept as antecedent. The syntax is

(**cl-remove-all-rules** <classified derived **CLASSIC** concept>)

When a rule is returned within LISP, it is printed as `@rule{name}`. There are a number of retrieval functions which operate on rules. These can be found in Appendix D.9.2.

Important note: see the end of Section 2.1.4.1 for a warning against the use of side effects within filler and description rule functions, as well as concept and individual definitions, and consequents of simple rules.

There is a library of test functions in the file library.lisp. These are functions which return one of 3 values: T, NIL, or ?. They are for use with test restrictions, mainly as filters for rules. When used appropriately, they will keep a rule from firing until all the necessary information is available. You can look through this file to see some example test functions. The function **cl-test-role-filled?**, defined above, is in this file. If you define any test functions which you think would be of general use, please mail a copy of them to resnick@research.att.com, to possibly include these functions in the library file.

2.7 Knowledge Bases

A knowledge base is simply a collection of concepts, individuals, roles, and rules, and the relationships between them.

3 Normalization, Subsumption, and Classification

Concepts and individuals are classified into hierarchies based on the *subsumption* relationship. Determining subsumption in CLASSIC is a two-part process, consisting first of normalization, and then of checking whether one normalized concept is more general than another. This section presents an informal discussion of this process, so you will understand how CLASSIC makes certain inferences, but it is not necessary for you to fully understand this section (especially the parts on classification) in order to be able to use CLASSIC.

3.1 Concepts

3.1.1 Normalization

When concepts are created, they first go through a process of *normalization* to get them into a normalized (or canonical) form, so that two concepts can be compared in a consistent manner, and questions of subsumption can be easily answered. Normalization involves:

- inheriting information from parent concepts,
- combining restrictions on the same role,
- eliminating embedded **AND** operators,
- deducing fillers when they are implied by **ALL** restrictions,
- deducing fillers when they are restricted to `INTEGER`s, and they are implied by a fully-specified interval (**MIN** and **MAX** restrictions are both specified),
- inheriting and combining role restrictions up and down role hierarchies,
- checking the consistency of restrictions, disjoint primitive concepts, etc.,
- combining **ONE-OF** restrictions (taking their intersection),
- filtering **ONE-OF** restrictions which are restricted to **HOST** individuals, by all applicable test restrictions, since the properties of **HOST** individuals cannot change (this is not done for **CLASSIC** individuals since their properties can change), and
- unifying **ALL** restrictions at the tips of **SAME-AS** restrictions.

For example, suppose a `HEALTHY-VEGETARIAN` is defined as

```
(AND PERSON
     (ALL food HEALTHY-THING)
     (ALL food PLANT))
```

where `PERSON`, `HEALTHY-THING`, and `PLANT` are all primitive concepts. Then the normalized form of `HEALTHY-VEGETARIAN` is

```
(AND PERSON
     (ALL food (AND HEALTHY-THING PLANT)))
```

The **ALL** restrictions on the `food` role were combined, and the embedded **AND**, which was redundant, was removed.

Now suppose we define an `OLD-HEALTHY-VEGETARIAN` as

```
(AND HEALTHY-VEGETARIAN
     (ALL age OLD))
```

Then the normalized form of OLD-HEALTHY-VEGETARIAN is

```
(AND PERSON
     (ALL food (AND HEALTHY-THING PLANT))
     (ALL age OLD))
```

It inherits the primitive superconcept PERSON, as well as the **ALL** restriction on food.

Suppose concept C is defined as

```
(ALL age (AND INTEGER
              (ONE-OF 20 25 30)
              (TEST-H evenp)))
```

The normalized form of C is

```
(ALL age (AND INTEGER
              (ONE-OF 20 30)
              (TEST-H evenp)))
```

Note that, since 25 is not an even integer, it has been filtered out.

The last part of normalization checks concepts for consistency. Examples of inconsistent concepts are:

- (**AND** (**AT-LEAST** 2 child) (**AT-MOST** 1 child));
- (**AND** (**ALL** child MALE) (**ALL** child FEMALE)), where MALE and FEMALE are two disjoint primitive concepts of the same disjoint grouping;
- (**AND** (**FILLS** child Jack) (**ALL** child (**ONE-OF** Mary Sue))), since Jack is not a member of the set (**ONE-OF** Mary Sue);
- (**AND** (**MIN** 3) (**MAX** 2)), an inconsistent interval;
- (**AND** (**FILLS** age 17) (**ALL** age (**MIN** 18))), since 17 is inconsistent with the specified interval.

3.1.2 Subsumption

A concept C_1 *subsumes* another concept C_2 if C_1 is an equivalent concept to C_2, or C_1 is a more general concept than C_2. Once two concepts have been normalized, we can easily determine whether one concept subsumes the other. Basically, in order for C_1 to subsume C_2, for each restriction on C_1 there must be an equivalent or more specific restriction on C_2.

For example, suppose we have the primitive concepts MAMMAL and PLANT under the concept CLASSIC-THING, the primitive concept PERSON under MAMMAL, and the primitive concept FRUIT under PLANT. Now we define the concept VEGETARIAN-MAMMAL as

```
(AND MAMMAL
     (ALL food PLANT))
```

and the concept FRUIT-EATING-PERSON as

```
(AND PERSON
     (ALL food FRUIT))
```

VEGETARIAN-MAMMAL subsumes (is more general than) FRUIT-EATING-PERSON, because:

- MAMMAL subsumes PERSON (PERSON was defined to be more specific than MAMMAL); and
- the **ALL** restriction on food for VEGETARIAN-MAMMAL, PLANT, subsumes the **ALL** restriction on food for FRUIT-EATING-PERSON, FRUIT.

3.1.3 The Classification Process

Classification of a concept involves finding the parents, children, and direct instances of the concept, and firing any rules with this concept as antecedent on all instances of the concept. We find the parents (most specific subsumers) of C as follows: start at the top of the concept hierarchy (with either CLASSIC-THING or HOST-THING, depending on the realm of C—assume it is a **CLASSIC** concept for the sake of this example). For each concept C_i that is known to subsume C (starting with CLASSIC-THING), check each of its children C_{i_j} to see if C_{i_j} subsumes C. The lowest such set of concepts in the concept hierarchy are C's parents.

To find C's children (most general subsumees), start with the parents you have just found for C, and for each child of the parents, see if C subsumes it. If so, you have found a child; if not, try its children until you have found a child of C, or a concept has no children.

To find C's instances (direct instances, not descendant instances), look at each individual which is a descendant instance of one of C's parents, but is not a descendant instance of one of C's children. If C subsumes it (see Section 3.2.2), it is an instance of C.

Now any rules with C as antecedent are fired on C's instances (descendant instances, i.e., all instances), and these individuals are reclassified (see Section 3.2.3 below).

3.2 Individuals

3.2.1 Normalization

The normalization of an individual includes all the steps involved in the normalization of a concept. In addition, it involves running any test functions on the individual (for checking its consistency), and propagating information to other individuals when new facts are implied by **ALL** restrictions and **SAME-AS** restrictions.

For example, suppose the concept FOOD-COMPATIBLE-PERSON is defined as someone whose favorite food is the same as his or her spouse's favorite food:

```
(AND PERSON
     (SAME-AS (favorite-food) (spouse favorite-food)))
```

Now suppose that we define Sally as

```
(AND FOOD-COMPATIBLE-PERSON
     (FILLS favorite-food Lasagne)
     (FILLS spouse Jim)
     (FILLS sister Mary)
     (AT-MOST 1 sister))
```

When Sally is normalized, she inherits the **SAME-AS** restriction from the concept FOOD-COMPATIBLE-PERSON, which causes Jim's favorite-food role to be filled with Lasagne. In addition, her sister role is closed, because she can have at most 1 sister, and that sister is known to be Mary.

After inheriting information from all its parents, an individual is checked for consistency. The examples of inconsistent concepts from Section 3.1.1 would also cause inconsistent individuals. The following are some additional examples of inconsistent individuals:

- John contains (**ALL** child MALE) and (**FILLS** child Mary), where Mary is a FEMALE, and MALE and FEMALE are two disjoint primitive concepts of the same disjoint grouping;
- Jack contains (**AT-LEAST** 2 child) and (**FILLS** child Jill) (and no other **FILLS** statements), and the child role is then closed with **cl-ind-close-role**;
- Sally has 2 fillers for the child role, Fred and Sam, and contains the restriction (**AT-MOST** 1 child);
- Harry contains (**ALL** age INTEGER) and (**FILLS** age 25.5).

3.2.2 Subsumption

An individual I is subsumed by, or satisfies, or is an instance of a concept C if I is described by C, i.e., if I satisfies every restriction on C. Once an individual has been normalized, we can easily determine whether or not it is subsumed by, or satisfies, a concept. Basically, in order for concept C to subsume a normalized individual I, each restriction on C must be satisfied by I, either because there is an equivalent or more specific restriction on I, or because the restriction on C can be derived from the restrictions, fillers, and closed roles on I.

For example, suppose we have the primitive concepts PERSON and ALCOHOLIC-BEVERAGE under the concept CLASSIC-THING, the primitive concepts WOMAN and DOCTOR under the concept PERSON, and the primitive concept WINE under the concept ALCOHOLIC-BEVERAGE (see Figure 2). Now we define the concept C_1 as a PERSON with at least 1 child, all of whose children are DOCTORS, and all of whose friends drink only ALCOHOLIC-BEVERAGES:

```
(AND PERSON
     (AT-LEAST 1 child)
     (ALL child DOCTOR)
     (ALL friend (ALL drink ALCOHOLIC-BEVERAGE)))
```

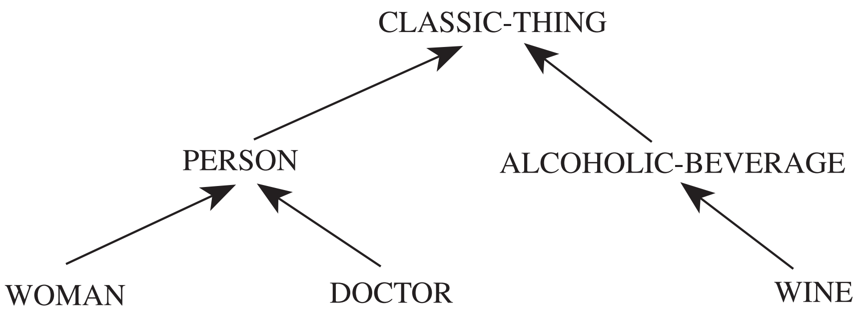

Figure 2: Hierarchy for individual subsumption example.

Now suppose that `Fran` is a `WOMAN` with exactly `2 children`, `Jack` and `Barbara`, both of whom have been previously asserted to be `DOCTORS`; all her `friends` drink only `WINE`; and `Susy` is one of her `friends`:

```
(AND WOMAN
     (FILLS child Jack Barbara)
     (AT-MOST 2 child)
     (ALL friend (ALL drink WINE))
     (FILLS friend Susy))
```

We know that $\mathtt{C}_1$ subsumes `Fran`, because each restriction on $\mathtt{C}_1$ is satisfied by `Fran`:

- `PERSON` subsumes `WOMAN`;
- Fran has *at least* `1 child`—in fact, she has `2 children` (and we know both of them);
- *all* her `children` are `DOCTORS` (we know all of them, and they have all been previously asserted to be `DOCTORS`);
- *all* her `friends drink` only `ALCOHOLIC-BEVERAGES`—in fact, they all `drink` only `WINE`, which is subsumed by `ALCOHOLIC-BEVERAGE`.

The extra fact we know about `Fran`—that `Susy` is her `friend`—does not affect this question of subsumption.

3.2.3 The Classification Process

Classification of an individual involves finding the parents of the individual, propagating any information from that individual to other individuals and reclassifying them, and firing any appropriate rules on the individual. The parents of an individual are found in the same way as the parents of a concept, using the subsumption check for individuals described above (an individual has no children, and of course, no instances).

If the individual has any new parents as a result of being classified, all rules with the new parents or their ancestors as antecedents are fired on the individual, and the individual is reclassified. For example, if `Mary` is classified as a `VEGETARIAN-PERSON`, then any rules with `VEGETARIAN-PERSON` or `PERSON` as antecedents are fired.

Information may be propagated from this individual to other individuals (see the example in Section 3.2.1 above, where `Jim`'s favorite food was inferred to be `Lasagne`). Any affected individuals are reclassified.

4 Incompleteness in CLASSIC

There are several types of incompleteness in CLASSIC which we must mention. For concept processing, subsumption is incomplete in several ways. Some of these incompletenesses also apply to individual processing. In addition, there are some incompletenesses in rule processing and propagations for individuals. Finally, for both concept and individual processing, there are certain inferences relating to role hierarchies which are not performed to completion. These types of incompleteness are all discussed below.

4.1 Concept Processing

There are a few ways in which concept subsumption is incomplete.

Test restrictions are treated as black boxes, in the same way as primitive concepts are, which means that the user must specify when one test restriction is more general than another, because CLASSIC does not figure it out. In fact there is really no way for CLASSIC to determine the meaning of test restrictions. For example, the built-in **HOST** concept `INTEGER` contains the test restriction (**TEST-H integerp**). If we define the concept `EVEN-INTEGER` as (**TEST-H evenp**), then it will *not* get classified under `INTEGER`, because CLASSIC does not know that anything which satisfies the **evenp** test must also be an `INTEGER`. Thus, we must tell CLASSIC that an `EVEN-INTEGER` must also be an `INTEGER` (see section 2.1.4.1). We would thus define `EVEN-INTEGER` as:

`(`**AND** `INTEGER (`**TEST-H evenp**`))`. See Section 9 for a description of some mechanisms for allowing CLASSIC to reason with test restriction (for advanced user only).

Also, with respect to concept subsumption, **CLASSIC** individuals are treated as having no properties. The reason for this is that we do not want the concept hierarchy to change when individuals change (concept definitions cannot change). This impacts the two operators **FILLS** and **ONE-OF**. If the individual `Mary` is known to be an `ATHLETE`, and concept `C` is the description of someone whose only child is `Mary`: `(`**AND** `(`**FILLS** `child Mary) (`**AT-MOST** `1 child))`, then `C` is *not* subsumed by the concept of someone, all of whose children are athletes: `(`**ALL** `child ATHLETE)`.

This applies to **CLASSIC ONE-OF** restrictions as follows: if we know that `Joe` and `Mary` are both `MAMMALS`, `(`**ONE-OF** `Joe Mary)` is not classified below `MAMMAL`. Thus, a **CLASSIC** concept containing only a **ONE-OF** restriction will be classified directly below either `CLASSIC-THING` or another concept containing only a **ONE-OF** restriction, but never below a concept containing any other type of restriction.

As a more complex example, suppose that `Susan` is known to have `Bob` as a `client` and `David` is known to have `Bill` as a `client`. Let `C1` be defined as

```
(AND COMPANY
      (AT-LEAST 1 employee) (ALL employee (FILLS client Jack))
      (ALL employee (ONE-OF Susan David))
      (ALL contractor (AT-MOST 1 client))
      (ALL contractor (ONE-OF Susan David)))
```

and let `C2` be defined as

`(`**AT-MOST** `1 contractor)`.

CLASSIC does *not* infer that C1 is subsumed by C2, because the establishment of the subsumption would require the use of contingent properties of Susan and David.

The above non-subsumptions are not really incompletenesses in CLASSIC, as the standard definition of subsumption ignores contingent properties of individuals. However, CLASSIC is incomplete with respect to this standard definition because it ignores properties of individuals that are implied by their presence in descriptions. For example, if C3 was defined as

```
(AND COMPANY
      (AT-LEAST 1 employee) (ALL employee (fills client Jack))
      (ALL employee (ONE-OF Susan David))
      (ALL contractor (AT-MOST 1 client))
      (ALL contractor (ONE-OF Susan David))
      (FILLS r Susan) (ALL r (FILLS client Bob))
      (FILLS s David) (ALL s (FILLS client Bill)))
```

CLASSIC would not infer that it was subsumed by C2, even though this inference does follow from the standard definition of subsumption.

To detect this subsumption, CLASSIC would have to determine that either Susan or David must be an EMPLOYEE; if Susan is an EMPLOYEE, then she can't be a CONTRACTOR because she has to have at least 2 clients; if David is an EMPLOYEE, then he can't be a CONTRACTOR because he has to have at least 2 clients. This reasoning by cases is computationally difficult, which is one reason we have chosen not to implement it in CLASSIC.

As another example, suppose that the concept C4 is defined as

```
(AND (FILLS child Sally) (ALL child ATHLETE)
      (FILLS friend Sally) (AT-MOST 1 friend))
```

and concept C5 is defined as (ALL friend ATHLETE). CLASSIC does *not* infer that C4 is subsumed by C5. As a final example, suppose that concept C6 is defined as

```
(AND (FILLS mother Sally) (FILLS best-friend Sally)
      (ALL mother (AT-LEAST 1 lawyer)))
```

and concept C7 is defined as

(SAME-AS (mother lawyer) (best-friend lawyer)).

CLASSIC does *not* infer that C6 is subsumed by C7.

HOST ONE-OF restrictions do not have the same types of incompleteness as **CLASSIC ONE-OF** restrictions, because the properties of **HOST** individuals do not change. A **HOST** concept containing only a **ONE-OF** restriction may be classified under a concept containing a test restriction, if all the **HOST** individuals in the **ONE-OF** restriction satisfy the test restriction. In addition, a **HOST** concept containing only a **ONE-OF** restriction can be classified under a concept containing an interval restriction, if the **ONE-OF** restriction contains only numbers, and they are all within the specified interval.

4.2 Individual Processing

The incompletenesses in concept subsumption can appear when determining whether or not an individual satisfies a concept description. This is because **ALL** restrictions on individuals are handled as concepts, not as descriptions of individuals. Suppose that the individual `Sam` is known to be a `VEGETARIAN`, and that the individual `Mary` is defined as someone all of whose friends have `Sam` as a teacher, and no one else:

(**ALL** `friend` (**AND** (**FILLS** `teacher Sam`) (**AT-MOST** `1 teacher`))).

`Mary` will not be found to satisfy the concept described by

(**ALL** `friend` (**ALL** `teacher VEGETARIAN`)),

because the properties of `Sam`, specifically that he is a `VEGETARIAN`, are not taken into account when doing the subsumption test.

Both rules and propagations are performed only on known instances. Thus, if we know that all `Mary`'s `sisters` are `ATHLETE`s, and she has at least 1 `sister`, we do not create a skolem individual representing the `sister` we know about, in order to reason about it.

Rules in CLASSIC are treated only as forward-chaining inferences, not as logical inferences. Thus, if we have a rule stating that if someone is a `VEGETARIAN`, then he is known to be a `HEALTHY-THING`, and we know that `Joe` is an `UNHEALTHY-THING` (a concept disjoint from `HEALTHY-THING`), we do *not* infer that `Joe` is not a `VEGETARIAN`.

4.3 Concept and Individual Processing

The following incompleteness applies to both concept and individual processing. For the two inferences **subrole-combination-implies-at-least** and **subrole-combination-implies-at-most** (see Section 7.4 for a description), a parent role and its subroles combine in certain ways to generate more specific **AT-LEAST** and **AT-MOST** restrictions. In certain cases, the "best" restriction is not calculated. Future versions of CLASSIC will perform these inferences more completely.

5 Updates

Once an individual has been created, it may be updated in one of two ways: additional information can be added to the individual, or existing information specified for an individual can be retracted. Both of these types of updates on individuals are described in this section. Rules and roles may not be modified (although rules *can* be removed). The definition of a concept also cannot be modified. The instances of a concept, however, can all get modified indirectly, if a rule is created which has that concept as the antecedent, because the rule will then fire on all instances of the concept. (See Section 2.6 for a description of rules.)

5.1 Addition

An individual can be asserted to have additional parents or restrictions, or new fillers for a role, with the **cl-ind-add** function. The syntax is

(**cl-ind-add** <derived individual> <**CLASSIC**-descr>)

where the individual is an existing individual, *not the name of an individual*, and the description is the information to be added to the individual, and is formed using the grammar in Appendix A. For example, to add the information that the existing individual `Mary` has `child John`, we would say:

(**cl-ind-add** `@Mary '(`**FILLS** `child John))` (the @ accesses the object with the given name (i.e., `Mary`)—see Appendix B).

It can also be asserted for an individual that the currently known fillers of a role are the only fillers (see **cl-ind-close-role** in Section 2.2.3 above). All information can be added to an individual using the two functions **cl-ind-add** and **cl-ind-close-role**, but for convenience, there are other functions for adding information to an existing individual (**cl-ind-add-concepts** and **cl-ind-add-fillers**)—see Appendix D.7.3.

5.2 Retraction

Information previously asserted about an individual can be retracted. **cl-ind-remove** retracts a **CLASSIC** description from an individual. For example, to retract the information that the existing individual `Mary` has `child John`, we would say:

(**cl-ind-remove** `@Mary '(`**FILLS** `child John)).`

cl-ind-remove-concepts retracts the information that an individual is described by a list of told concepts. For example, to retract the information that the existing individual `Mary` is an `ATHLETE`, we would say:

(**cl-ind-remove-concepts** `@Mary (`**list** `@tc{ATHLETE})).`

cl-ind-remove-fillers removes a list of told fillers for a particular role on an individual. **cl-ind-remove-all-fillers** removes all told fillers for a particular role on an individual. **cl-ind-unclose-role** removes the told fact that a particular role is closed on an individual. See Appendix D.7.4 for a complete description of these functions.

5.3 Batch Processing

In the normal processing mode, whenever an update function is called on an individual, CLASSIC immediately performs all the inferences on that individual and any affected individuals. Suppose we have previously defined the concepts `VEGETARIAN` (a `PERSON` who eats only `PLANT`s) and `PARENT-OF-VEGETARIAN` (a `PERSON` all of whose `children` are `VEGETARIAN`s), and we have also told CLASSIC that `Mary`'s only `child` is `John`. Suppose we then do an addition: we tell CLASSIC that the only kind of `food John` eats is of type `PLANT`. CLASSIC will immediately classify `John` as a `VEGETARIAN`, and then it will classify `Mary` as a `PARENT-OF-VEGETARIANS`. Now suppose that at some later time we do a retraction: we remove the assertion that the only kind of `food John` eats is of type `PLANT`. This would cause CLASSIC to immediately undo the inferences it had made earlier, and it would no longer contain the information that `John` is a `VEGETARIAN`, and that `Mary` is a `PARENT-OF-VEGETARIANS`.

There is a special batch processing mode, in which CLASSIC delays all processing of new information until the batch mode is exited. This allows the processing to be much more efficient in the case where some individuals are affected by many different updates. CLASSIC currently has batch processing implemented only for retraction. Thus, in the example above, we could go into batch retraction mode, retract the information about `John` and perhaps other pieces of information that might affect `John`, and then exit batch retraction mode, which would cause CLASSIC to process all modified and affected individuals at one time. CLASSIC may have batch processing implemented for additions at a later date.

The variable ***cl-system-status*** is bound to a symbol which is the mode the system is currently in: either **:normal** (non-batch processing—the default), or **:batch-retraction** (currently doing a batch retraction).

The function **cl-start-batch-retraction** starts a batch retraction. It can only be called if ***cl-system-status*** is **:normal**. It resets ***cl-system-status*** to **:batch-retraction**. All individual retraction calls done while in **:batch-retraction** mode are recorded but not processed. While in **:batch-retraction** mode, no addition functions may be called (e.g., **cl-ind-add**). Also, no functions may be called which access derived information about any **CLASSIC** individuals (e.g., **cl-print-ind** and **cl-primitives** when called on a **CLASSIC** individual, or **cl-concept-instances**), because any individual may be in a partially processed state. If any of these types of functions is called, a local error will be generated, but this will not affect batch processing—batch retraction resumes as soon as the error has been reported. Told information may be accessed while in **:batch-retraction** mode.

The function **cl-end-batch-retraction** ends a batch retraction. It can only be called if ***cl-system-status*** is **:batch-retraction**. It causes all retractions and inferences to be performed for any retractions done since **cl-start-batch-retraction** was called, it checks all affected individuals for consistency, and then it resets ***cl-system-status*** to **:normal**.

If an inconsistency error is detected during the processing of a **cl-end-batch-retraction**, all the updates that were processed during the batch retraction are undone, ***cl-system-status*** is reset to **:normal**, and the knowledge base reverts to the state it was in before the **cl-start-batch-retraction** was called. If an inconsistency error is detected, it is guaranteed that an error would have been found if the same list of retraction functions were called in non-batch mode. However, it is possible that a list of functions that would lead to an error in non-batch mode will not produce an error in batch mode. For example, suppose the `age` role is closed on `John`. The following would produce an error in non-batch mode, but would not produce an error in batch mode:

```
(cl-ind-remove-fillers @John @age '(22))
(cl-ind-unclose-role @John @age)
```

See Section 6 to see how batch processing affects the list ***cl-error-object-list***, which is generated when an error is detected.

6 Error Handling

This section discusses the handling of errors and the printing of debugging and warning messages in CLASSIC.

The setting of the CLASSIC error mode determines how an error will be handled. If the error mode is on, then when a CLASSIC error is signaled, "*CLASSIC ERROR* while running in error mode" is printed. CLASSIC then puts the user into the LISP debugger, so she can peruse the stack and get more information about where the error occurred.

If the CLASSIC error mode is off (the default), then when a CLASSIC error is signaled, "*CLASSIC ERROR* while processing <form>" is printed (where <form> is the form that caused the error) followed by an appropriate error message, and multiple values are returned. The first return value is the symbol `classic-error`, and the second return value is a list. The first element of the list is an error code, and the rest of the list contains the arguments to the error (if any), i.e., the context of where the error occurred. (The information contained in the error code and arguments is reflected in the error message that is printed.) For example, suppose you try to create the individual `Mary` as follows:

(**cl-create-ind** `'Mary '(`**AT-LEAST** `1))`,

where you forget to specify the role name for the **AT-LEAST** restriction. An error message will get printed:

> *CLASSIC ERROR* while processing
> (**cl-create-ind** `Mary (`**AT-LEAST** `1)):`
> Incorrect number of arguments in classic expression: `(`**AT-LEAST** `1).`

CLASSIC returns the symbol `classic-error` and the list

(INCORRECT-NUMBER-OF-ARGS-ERROR (**AT-LEAST** 1)).

For parsing (syntax) errors (for example, an object is of the wrong type or a CLASSIC operator has an incorrect number of arguments), only `classic-error` and the list of error code and arguments are returned. However, for "real" errors—those where an inconsistency is found while creating an object—there are two more return values. The third return value is a list of roles, which is the role-path where the error occurred (the error role-path). The fourth return value is a derived concept or individual which contains all the inconsistent information which led to the error state. We will call this object an *error object.*

The CLASSIC predicates and printing, retrieval, and explanation functions may all be used on an error object. However, this object is not in the knowledge base, so none of the creation or addition functions may use it. For example, an error concept cannot be the antecedent of a rule, and an error individual may not be added as the filler of a role. *Do not try to use error objects in this way!*

An error object prints out with the name it was defined with, but this name cannot be used to retrieve the object from the knowledge base (i.e., with **cl-named-ind** or **cl-named-concept**). When an error individual is returned, it is printed as @i{<name>`-*INCOHERENT*`}, e.g., @i{`MARY-*INCOHERENT*`}, and similarly, an error concept is printed as @c{<name>`-*INCOHERENT*`}. An error rule concept is printed as @c{`RULE-CONCEPT-*INCOHERENT*`} (the name is the symbol `RULE-CONCEPT`). (See below for a description of error rule concepts.)

The following is an example of a "real" CLASSIC error. Suppose we type

```
(cl-create-ind 'Mary
               '(AND (AT-MOST 0 son)
                     (FILLS son Jack)))
```

(assuming the `son` role has already been defined). Then along with some warning messages, we get the following error message:

> *CLASSIC ERROR* while processing

(**cl-create-ind** Mary (**AND** (**AT-MOST** 0 son) (**FILLS** son Jack)))
occurred on object @i{Mary-*Incoherent*}
along role-path (@r{son}):
Inconsistent number restriction for role @r{SON}—at-least:1; at-most 0.

CLASSIC returns the symbol classic-error, the list
(INCONSISTENT-BOUNDS-CONFLICT @r{SON} 1 0),
the role-path (@r{son}), and an error individual which looks like
@i{MARY-*INCOHERENT*}.

To write LISP code that correctly handles potential CLASSIC errors, you should surround your calls to CLASSIC functions with tests for the first return value (i.e., using **multiple-value-bind**). If the first return value is the symbol classic-error, you should handle the error in an appropriate way based on the error information contained in the second return value. (You can also call the explanation functions on the error object—see Section 7.3.) If there is no error, the return value(s) will be the expected return value(s) of the CLASSIC function that was called. See below for a way to avoid using **multiple-value-bind**. See Appendix E for a description of all CLASSIC error codes.

When a "real" error is encountered in CLASSIC (i.e., one which generates an inconsistent object, as described above), not only is an error object returned as the fourth value, but in addition, a copy of this error object, and copies of all individuals which were modified as a result of the problematic input are stored on the list ***cl-error-object-list***. Each individual on this list contains information derived from any inferences performed based on the problematic input. Exactly one object on this list will be a real *error object*, e.g., it will print out as @i{MARY-*INCOHERENT*}. This is the object where the actual conflict occurred. The other individuals on the list, which are just copies of real individuals, will print out as @i{<name>-*COPY*}, e.g., @i{JOHN-*COPY*}. However, we will use the term *error object* to refer to all the objects on this list. The objects on this list can be used to trace the source of each piece of information which caused an error. This is described in Section 7.3.

Whenever a new input causes an error, ***cl-error-object-list*** will point to a new list containing the new error individuals. Thus, the list of error individuals from the previous error will no longer be accessible, unless you have saved it as the value of a variable. (This list will also get set to NIL when **cl-startup** is called.)

The function **cl-error-object** takes the name of an error object, and an optional object type (**concept** or **individual**) and list of error objects (defaults to ***cl-error-object-list***), and returns the error object in the list which has the given name and type, or it returns the first object in the list with the given name, if no type is specified. For example, to access the error individual named Mary, you would type
(**cl-error-object** 'Mary).
Most of the error objects will be individuals. The only exception is for the object which actually contains the inconsistency. This can be an individual or a concept, where the concept is just the concept that the user was creating, or an "error rule concept." An error rule concept is generated if a rule consequent is in conflict with an antecedent concept, or with other rule consequents associated with a single antecedent concept. The rule concept contains all the information from the antecedent concept and all rule consequents that are associated with the antecedent and have no filter. As mentioned above, an error

rule concept is printed as @c{RULE-CONCEPT-*INCOHERENT*}. The name of an error rule concept, for the purpose of calling **cl-error-object**, is the symbol RULE-CONCEPT. Thus, to retrieve an error rule concept from the list ***cl-error-object-list***), you would type

(**cl-error-object** 'RULE-CONCEPT).

Note that for interactive uses of CLASSIC, you may ignore the error object returned as the fourth return value, since there will be an equivalent object on the list ***cl-error-object-list***. Thus, you can avoid using **multiple-value-bind**, and just use **cl-error-object** to retrieve the appropriate object. However, the fourth return value is available in case you want a direct handle on the particular object which contains the inconsistency (e.g., in case your application does some automatic processing on this object).

Note that if a "real" error is encountered during batch processing (e.g., after calling **cl-end-batch-retraction**—see Section 5.3 for a description of batch processing), only the local object which caused the error is put onto ***cl-error-object-list***. This way CLASSIC avoids generating a huge list, consisting of copies of all individuals which were touched during batch processing. If you wish to have access to all the individuals involved in producing the error (perhaps to call explanation functions on them—see Section 7.3), you may re-execute all the commands that were done in `:batch-retraction` mode, but this time execute them in `:normal` mode (i.e., don't call **cl-start-batch-retraction**). Only the individuals which were modified as a result of the single input that caused the error are then copied and stored on ***cl-error-object-list***.

See Appendix D.3.1 to see how to set up error hook functions—functions which will get executed every time an error is encountered in CLASSIC.

The function **cl-set-classic-error-mode** turns the CLASSIC error mode on or off. The syntax is

(**cl-set-classic-error-mode** &optional (<boolean> NIL))

It takes an optional boolean argument (the default is NIL), and turns the CLASSIC error mode on for a non-NIL value or off for NIL.

The function **cl-classic-error-mode** returns T or NIL, depending on whether the CLASSIC error mode is currently on or off, respectively.

The error messages are all printed to the stream that ***cl-error-msg-stream*** is bound to (the default is the value of ***standard-output***)—thus the output can be redirected to a stream of the user's choice.

The function **cl-set-classic-warn-mode** turns the CLASSIC warning mode on or off. The syntax is

(**cl-set-classic-warn-mode** &optional (<boolean> T))

It takes an optional boolean argument (the default is T), and turns the CLASSIC warning mode on for a non-NIL value or off for NIL. When the CLASSIC warning mode is on (the default), CLASSIC will warn the user of certain events which may indicate that she has done something wrong; for example: when a new individual is being created as the result of being inside a **ONE-OF** description (she might have spelled the individual's name incorrectly); when the **ALL** restriction on a role is an incoherent concept (it can have no instances, i.e., if someone's `children` eat only MEAT, and they also eat only VEGETABLES, and MEAT and VEGETABLES have been declared to be disjoint, then that person can have no `children`—this might happen as the result of a user's mistake). If a human user is interacting with the knowledge base, it is suggested that the CLASSIC warning mode be turned on. If the knowledge base is stable, however, the user may elect to turn off the warning mode. When CLASSIC is interacting with a program, the warning mode should be turned off, unless a record of all the warning messages printed is being saved in a file.

The function **cl-classic-warn-mode** returns `T` or `NIL`, depending on whether the CLASSIC warning mode is currently on or off, respectively.

The warning messages are all printed to the stream that ***cl-warn-msg-stream*** is bound to (the default is the value of ***standard-output***)—thus the output can be redirected to a stream of the user's choice.

The user can also program new warning messages to be printed from her application with the function **cl-print-warn**. The syntax is

(**cl-print-warn** <string> <arg>*)

It takes a control string to print out, and some arguments to that string, in the same form as the `format` function. When the CLASSIC warning mode is turned on (the default), prints out the format string on the stream to which ***cl-warn-msg-stream*** is bound. It always returns `T`.

The function **cl-set-classic-debug-level** sets the debugging level for CLASSIC, which specifies how much helpful information is printed while running CLASSIC. The syntax is

(**cl-set-classic-debug-level** &optional (<level> `1`))[14]

The user may set the debugging level to any non-negative integer, although if the `level` argument is unspecified, it defaults to `1`. For the built-in CLASSIC debugging messages, the following conventions are used for the debugging levels:

- `0`: no debugging information is printed. This is the initial debugging level, as well as the debugging level after **cl-startup** has been called (see Appendix D.1.2). It is useful when a program, and not a user, is interacting with CLASSIC, or when the user has completely debugged a knowledge base;

- `1`: some error information is printed when the creation of a rule is unsuccessful;

- `2`: a message is printed when a rule is created successfully.

The function **cl-classic-debug-level** returns the current debugging level.

The debugging messages are all printed to the stream that ***cl-debug-msg-stream*** is bound to (the default is the value of ***standard-output***)—thus the output can be redirected to a stream of the user's choice.

The user can also program new debugging messages to be printed from her application, with the function **cl-print-debug**. The syntax is

(**cl-print-debug** <level> <string> <arg>*)

It takes a level (a positive integer), a control string to print out, and some arguments to that string, in the same form as the `format` function. When the current debugging level is greater than or equal to the level argument, it prints out the format string to the steam that ***cl-debug-msg-stream*** is bound to. It always returns `T`.

[14] The &optional keyword, used in function headers, says that the argument following is optional. If the argument is a two-element pair surrounded by parentheses (), the first element is the argument type and the second element is its default value, used when no argument is given.

To avoid printing any of the types of messages described in this section, the user can set the appropriate stream variable (described above) to ***cl-null-stream***, which is the null stream. Any output sent to this stream will be discarded.

7 Explanation

CLASSIC version 2 has a facility that allows the user to ask for an explanation of anything that CLASSIC has derived about a CLASSIC object (concept, individual, or consequent of a simple rule). Conceptually[15], the explanation functions begin with the object to be explained and "throw out" the derived information about that object (while maintaining the original "told information"). Then the information about the object to be explained is rederived beginning with the told information, but this time an extensive annotation is maintained which captures the inferences that were done and the other objects that were used to derive the new information. Explanation objects are stored on all pieces of information which are to be explained. An explanation object consists of an inference name (for a list of the inferences, see Section 7.4) and the arguments that were needed to perform this inference. These arguments are other CLASSIC objects. For example, a piece of information on object `A` may have been inherited from a parent concept `B` at the top level (not along any role-path). Thus there would be an explanation object associated with this piece of information which contains the inference name, **inheritance**, and the arguments, which are the parent concept `B`, and `NIL` (for the role-path along which `B` was found). If the piece of information on `A` was derived in multiple ways, then it would have multiple explanation objects, which contain information about each way that information was derived. CLASSIC takes a lazy approach to explanation generation in that it only produces explanation structures on demand and it only generates the last step in any explanation chain (i.e., it would not produce an explanation object in the previous example about how the piece of information in question came to be on concept `B`. If the user wants that information, another explanation request would be made on concept `B`, or a call to **cl-complete-exp** can be made on object `A`.) This allows CLASSIC to limit the amount of information that is displayed as an answer to any single explanation request.

7.1 Explanation Functions

There are two different kinds of explanation functions: normalization explanation functions and subsumption explanation functions. The normalization ones are described first. Note that once a subsumption explanation function has been run on an object, that object may contain certain information specific to subsumption, so that when the normalization printing or retrieval functions are called, you may see some subsumption information not described in Section 7.1.1. These differences are discussed at the beginning of Section 7.1.2. The explanation inferences are described in Section 7.4.

[15]In reality, the functions are written to be more efficient and to do a static analysis of the existing CLASSIC objects wherever possible in order to avoid doing recomputation of derived information

7.1.1 Normalization Explanation Functions

The following functions are used to set up, clear, print, and extract explanation information on an object. The functions described here which do any setting-up of explanation (**cl-exp**, **cl-complete-exp**, and **cl-set-up-exp**) just calculate the normalization explanation information for the object. The normalization explanation inferences (those calculated by these functions) explain where a particular piece of information on an object came from. These inferences are described in Section 7.4.1. The other functions in this section extract or print all explanation information which has been calculated, including the subsumption explanation (see Section 7.1.2 for functions to calculate subsumption explanation).

cl-exp

The main explanation function is **cl-exp**. The syntax is

(**cl-exp** <object> &key (role-path `T`) (aspect `T`) (aspect-filler `T`) (all `NIL`))

It takes a classified derived concept (it cannot take something which prints as @c{`UNCLASSIFIED-CONCEPT`}), a derived **CLASSIC** individual, or a simple rule. (For a simple rule, it explains the information in the rule consequent. This also applies to **cl-complete-exp**, **cl-clear-exp**, **cl-set-up-exp**, **cl-get-exp**, and **cl-print-exp**.) First it clears out any pre-existing explanation structures for the object. Then it calculates the explanation for that object, and finally, it prints out the explanation it calculated. If role-path is a list of roles or a single role, then just the explanation along this role-path will be calculated. If role-path is `T` (the default), **cl-exp** will calculate explanation along all role-paths. If aspect is a valid aspect (**ONE-OF**, **TEST**, **PRIM**, **MIN**, **MAX**, **AT-LEAST**, **AT-MOST**, **ALL**, **FILLS**, or **SAME-AS**), then just the explanation of this aspect will be calculated. If aspect is `T` (the default), **cl-exp** will calculate explanations for every aspect. For the aspects **FILLS**, **PRIM**, **TEST**, **ONE-OF**, and **SAME-AS**, if aspect-filler is `T`, all the objects for the specified aspect are explained; otherwise, aspect-filler specifies which filler, primitive, test, **ONE-OF**, or **SAME-AS** restriction to explain. For **FILLS**, the aspect-filler must be a derived **CLASSIC** individual or a **HOST** individual. Although **PRIM** is not a constructor it may be used to explain why a primitive derived concept is a primitive ancestor of a given concept. When the **PRIM** aspect is used the role-path value is ignored and the aspect-filler must be a primitive derived concept. For **TEST**, the aspect-filler must be a list containing a function and optional arguments. For **ONE-OF**, the aspect-filler must be a list of derived **CLASSIC** or **HOST** individuals, or the symbol **final-one-of**. If it is **final-one-of**, then the final **ONE-OF** restriction which has been derived on the object will be explained. For **SAME-AS**, the aspect-filler must be a list of 2 attribute paths. If an aspect-filler is specified for any other aspect, or `T`, or the aspect filler is not legal for the aspect, the error ILLEGAL-ASPECT-FOR-ASPECT-FILLER-ERROR will be generated. If the all keyword is `NIL` (the default), then the printing function will not print out explanations for anything where the explanation is unknown. If it is non-`NIL` and a piece of information does not have a known source, then "unknown" will be printed.

Note that **cl-exp** has the effect of printing the explanation for an object for a single role-path or all role-paths, a single aspect or all aspects, and a single aspect-filler or all aspect-fillers. If you wish to have the effect of printing the explanation along more than one specified role-path, aspect, and/or aspect-filler, you can first call **cl-clear-exp**, then call **cl-set-up-exp** several times with the appropriate role-paths, aspects, and aspect-fillers, and finally, call **cl-print-exp** to print out all the explanation information you have calculated. (See the descriptions of these functions below.)

As an example of using **cl-exp**, we could print an explanation of the **AT-LEAST** restriction for `Mary`'s `children`'s `friends` as follows:

(**cl-exp** @`Mary` :role-path (list @`child` @`friend`) :aspect '**AT-LEAST**)

We could print an explanation of why `Bill`'s `friend` is `Bob`, as follows:

(**cl-exp** @`Bill` :role-path (list @`friend`) :aspect '**FILLS** :aspect-filler @`Bob`)

cl-complete-exp

The function **cl-complete-exp** is like **cl-exp**, except that it traces the explanation of a fact all the way back to its source. Thus, if the individual `Mary` is an `ATHLETE` because `Sue`'s `child` is `Mary`, and `Sue` contains the restriction `(`**ALL** `child ATHLETE)`, then complete explanation will also explain how `Sue` got this restriction.

The syntax is

(**cl-complete-exp** <object> &key (role-path `T`) (aspect `T`) (aspect-filler `T`) (all `NIL`))

All the explanation information is calculated on all relevant objects, and then this information is printed out one object at a time (as if **cl-print-exp** were called on each affected object).

Note: normally, after complete explanation is run on an object, the explanation structures on the object contain the cumulative explanation information that was printed for the complete explanation. However, when a piece of information is explained by a rule firing, and the rule firing is then explained, the information in the individual's explanation structures will be wiped out before the rule firing is explained, so the information left in the explanation structures after calling **cl-complete-exp** may not be what you expect. This wiping out of information and explanation of the rule firings can be avoided if you turn on the pruning variable ***cl-complete-exp-ignore-rules*** (see Section 7.2).

cl-clear-exp

The function **cl-clear-exp** clears all the explanation structures in a derived concept, derived **CLASSIC** individual, or simple rule. The syntax is

(**cl-clear-exp** <object>)

cl-set-up-exp

The function **cl-set-up-exp** sets up the explanation for a classified derived concept, a derived **CLASSIC** individual, or a simple rule. (It is different from **cl-exp** in that it does not first clear the explanation structures, and after calculating the explanation it does not print it out.) The syntax is

(**cl-set-up-exp** <object> &key (role-path `T`) (aspect `T`) (aspect-filler `T`))

The role-path, aspect, and aspect-filler keywords have the same meaning as in **cl-exp**.

cl-get-exp

The function **cl-get-exp** takes a derived concept, derived **CLASSIC** individual, or simple rule, a role-path, an aspect, and an optional argument (an aspect-filler), and returns a list of explanation objects and a boolean value. The syntax is

(**cl-get-exp** <object> <role-path> <aspect> &optional (<aspect-filler> T))

It follows the role-path along the object, and returns (as the first return value) a list of explanation structures which explain the specified aspect. For a simple rule, it explains the rule consequent. Each explanation structure returned is printed as <exp>, and can be examined with the functions **cl-exp-type** and **cl-exp-args**. The second return value is T if the function correctly followed the role-path, and NIL otherwise. Thus, if the role-path exists on the object, but there is no explanation object stored for the particular aspect, then NIL,T is returned.

For the aspects **FILLS**, **PRIM**, **TEST**, **ONE-OF**, and **SAME-AS**, the optional aspect-filler specifies which filler, primitive, test, **ONE-OF**, or **SAME-AS** restriction to return the explanation for. If the aspect-filler is T (the default), then a list of lists will be returned, where for each inner list, the car is the particular filler, primitive, test, or **SAME-AS** restriction explained, and the cdr is a list of explanation objects explaining it. For the **ONE-OF** aspect, the aspect-filler can be **final-one-of**, as described in **cl-exp**. Note that if a subsumption function has been used to set up explanation on an object, then one of the five symbols **filler-list**, **prim-list**, **test-list**, **one-of-list**, or **same-as-list** may be used as the aspect-filler, or may be returned when the aspect-filler is T. See the beginning of Section 7.1.2 for a discussion of these symbols.

As an example of using **cl-get-exp**, suppose Mary's children are Bob and Fred, and we want the system to explain where the filler Bob came from. First we must set up the explanation on Mary:

(**cl-set-up-exp** @Mary)

If we then type

(**cl-get-exp** @Mary (list @child) '**FILLS** @Bob)

the system returns a list containing a single explanation object, which is printed as (<exp>). We can then look at the type and arguments of the explanation object with the functions **cl-exp-type** and **cl-exp-args**, respectively (see below).

If we type

(**cl-get-exp** @Mary (list @child) '**FILLS**)

the system returns something that looks like

((@i{Bob} <exp>) (@i{Fred} <exp>))

cl-print-exp

The function **cl-print-exp** takes a derived concept, derived **CLASSIC** individual, or simple rule, and prints out the explanation for the specified role-path, aspect, and aspect-filler. The explanation must have been previously set up by **cl-exp**, **cl-complete-exp**, or **cl-set-up-exp**, or one of the subsumption explanation functions. **cl-print-exp** does not calculate the explanation—it just prints the explanation information which is already there. The syntax is

(**cl-print-exp** <object> &key (role-path T) (aspect T) (aspect-filler T) (all NIL))

The printing of explanation information on an object (using **cl-print-exp**) is similar to the regular printing of the object (using **cl-print-object**). However, for explanation, after a piece of information is printed, each of the inference types which explains the information, along with its optional arguments, is printed. Sometimes the arguments are just listed:

INHERITANCE: (@tc{WOMAN} @tc{ATHLETE});
and sometimes they are shown descriptively:
PROPAGATION: from individual: @i{MARY}; role: @r{child}.
In any case, the actual arguments can be retrieved using **cl-get-exp** and **cl-exp-args**. (See Section 7.4 for a description of the inference types and their arguments.)

When there is a **SAME-AS** restriction in the object being explained, and the role-path is T, the complete restrictions are explained on only one of the role-paths. When the end of the second role-path is reached, CLASSIC prints:

Equivalent by SAME-AS to role-path: (<first role-path>),

where <first role-path> is the role-path which was explained. This is to avoid going into an infinite loop when printing recursive **SAME-AS** restrictions such as (**SAME-AS** (r1) (r1 r2)).

Sometimes an inference will show up on a different role-path of the **SAME-AS** restriction than the one it came from. In addition, information at the end of either or both role-paths may show either the **same-as** inference or one of the **same-as-implies-** inferences, when in fact, the inference only applies to one of the role-paths. (CLASSIC does not distinguish between these identical role-paths.) These problems will be fixed in a future version of CLASSIC.

Note that if a subsumption function has been used to set up explanation on an object, then **Subsumption:** and **Origin:** may show up in the printing of an explanation, when **cl-print-exp** is called. See the beginning of Section 7.1.2 for a discussion of this.

cl-exp-type

The function **cl-exp-type** takes an explanation object (as returned by **cl-get-exp**), and returns the inference type of the explanation. These inference types are described in Section 7.4. The syntax is
(**cl-exp-type** <explanation-object>)
Note that **cl-get-exp** returns a list of explanation objects, so we must call **cl-exp-type** on a single element of that list.

cl-exp-args

The function **cl-exp-args** takes an explanation object (a list of explanation objects is returned by **cl-get-exp**), and returns the arguments of the explanation. The arguments for each inference type vary. They are explained in detail in Section 7.4. The syntax is
(**cl-exp-args** <explanation-object>)

Consider an example where the concept C1 is defined as someone whose child role is filled with Bob. Then suppose we create Mary as a C1. Now we want to get the explanation type and arguments for this filler of the child role on Mary. First, we must set up the explanation:
(**cl-set-up-exp** @Mary :role-path (list @child) :aspect '**FILLS**)
Now we can get the explanation object for this filler as follows:
(**setq** exp-object (**car** (**cl-get-exp** @Mary (list @child) '**FILLS** @Bob)))
Then we can get the type and arguments for the explanation object:

(**cl-exp-type** exp-object) returns **INHERITANCE**, and

(**cl-exp-args** exp-object) returns (@tc{C1} NIL). Note: in this example, the first argument, @tc{C1} is the told concept from which the information was inherited, and the second argument, NIL, is the role-path along which C1 was found, i.e., it was at the top level.

7.1.2 Subsumption Explanation Functions

A concept C_1 *subsumes* another concept C_2 if and only if C_1 is equivalent to C_2, or C_1 is a more general concept than C_2. More specifically, in determining whether a concept C_1 subsumes C_2 we look at every piece of information on C_1 and we must find an equivalent or more specific piece of information on C_2 (see Section 3.1.2 for a more detailed discussion of concept subsumption, and Section 3.2.2 for a detailed discussion of individual subsumption). The functions described in this section provide explanations for why one object subsumes another and in particular, for each piece of information on the subsuming concept we explain why the corresponding piece of information on the subsumed concept is at least as specific as the information on the subsuming concept. For example consider the following two concepts:

```
C1= (AND PERSON
         (AT-LEAST 1 sister)
         (ALL sister PROFESSIONAL))

C2= (AND PERSON
         (AT-LEAST 3 sister)
         (ALL sister DOCTOR)
         (ALL friend (ALL drink ALCOHOLIC-BEVERAGE)))
```

If we ask the system to explain why C1 subsumes C2 it will return the following information which is presented in a slightly edited form:

```
primitives:  (PERSON CLASSIC-THING)
             PRIMITIVE-SUBSET
Role Restrictions:
     SISTER
           at-least (3):
           AT-LEAST-ORDERING: At-least:3 satisfies at-least:1
     all DOCTOR:
         SUBSUMING-ALL-RESTR: PROFESSIONAL subsumes DOCTOR.
```

This information first states that the set of primitives on C2 is at least as specific as the primitives on C1 since the list of primitives on C1={PERSON,CLASSIC-THING} is a subset of the set of primitives on C2={PERSON,CLASSIC-THING}. It indicates that the **AT-LEAST** restriction on C2 is more specific than the **AT-LEAST** restriction on C1, and that the **ALL** restriction on the sister role is more specific on C2 since PROFESSIONAL subsumes DOCTOR. Note that what C2's friends drink is ignored since there is no corresponding constraint on

C1 and extraneous information on the subsumed concept is ignored since it can only make the concept more specific.

cl-exp-subsumes-concept

The function **cl-exp-subsumes-concept** takes two derived concepts: a subsuming concept (the subsumer) and a subsumed concept (the subsumee), and explains why the subsuming concept *subsumes* the subsumed concept. The function generates an error if the first concept does not in fact subsume the second concept. The syntax is

(**cl-exp-subsumes-concept** <derived concept> <derived concept>
&key (reason T) (origin NIL) (all NIL))

The origin keyword when set to true provides additional explanation about the origin of the information on the subsumee which is directly involved in the subsumption relation. Thus in the example from above, setting the origin keyword to true would explain how PERSON came to be a primitive parent of C2 as well as explaining the origin of the **AT-LEAST** and **ALL** restrictions on the sister role. In this case the origin of all the information would be explained to result from the **told-info** inference. Note that the **ALL** restriction on what C2's friends drink would not have its origin explained since it is not relevant for the subsumption being queried. Although we have omitted this from the printout of the example above for simplicity, all subsumption inferences are preceded with **Subsumption:** and all explanations of the origin of information are preceded with **Origin:**.

The reason keyword by default is set to true. It causes the explanations of the subsumption inferences to be generated. If both the reason and origin keywords are set to NIL then **cl-exp-subsumes-concept** will not do anything. The only practical use of the non-default setting of the reason keyword to false is if the origin keyword is set to true, in which case the origin of all information on the subsumee relevant to the subsumption would be explained, but the reasons why the information is at least as specific as the corresponding information on the subsumer would **not** be explained.

The use of the all keyword determines what information is involved in explaining a given subsumption relationship. Normally when we are explaining a subsumption we want to explain why all the told information on the subsumer is more general than the conjunction of the told and the derived information on the subsumee. When the all keyword is set to true we explain why all the told **and** derived information on the subsumer is satisfied by the information on the subsumee. In the example above, if the all keyword were set to true we would explain how the primitive parent PERSON and the derived primitive parent CLASSIC-THING are satisfied by C2, whereas if all is set to false we only explain how PERSON is satisfied by C2 since the fact that CLASSIC-THING is a parent of C1 is derived information. The effect of this is clearest when the origin keyword is also set to true, for example if both are set to true in explaining why C1 subsumes C2 then it would explain the origin of the primitive parent CLASSIC-THING on C2 as well as the origin of the primitive parent PERSON. The reason that the all keyword is non-intuitive is that in order for C1 to subsume C2 it is only necessary for C2 to satisfy the told information on C1 since if it does it will automatically acquire all the derived information that `C1` did and possibly more.

cl-exp-subsumes-ind

The function **cl-exp-subsumes-ind** takes a derived concept and a derived individual, and explains why the concept *subsumes* the individual, i.e., why the individual is an instance of the concept. The syntax is

(**cl-exp-subsumes-ind** <derived concept> <derived individual>
&key (reason `T`) (origin `NIL`) (all `NIL`))

The keywords operate the same for this function as for **cl-exp-subsumes-concept**, and it also generates an error if the concept does not subsume the individual. This is the only explanation function that can take a **HOST** individual. Since a **HOST** individual is just the Common LISP object (string, number, etc.), CLASSIC cannot actually store the explanation information it calculates on the individual. However, it will calculate this information and print it out.

As an example of using this function, suppose that concept `C1` is defined as someone whose `mother` is the same as her `best-friend`:

(**cl-define-concept** `'C1 '(`**SAME-AS** `(mother) (best-friend)))`,

where `mother` and `best-friend` are attributes. Now create the individual `Mary` as someone whose `mother` and `best-friend` are both filled with the individual `Sue`:

(**cl-create-ind** `'Mary '(`**AND** `(`**FILLS** `mother Sue)`
`(`**FILLS** `best-friend Sue)))`

If we call (**cl-exp-subsumes-ind** `@C1 @Mary` :origin `T`), we see that the **SAME-AS** restriction on `C1` was satisfied (by subsumption) by the inference **satisfies-same-as-with-filler**, and the filler is `Sue`. We then see that the origin of the filler `Sue` for both the `mother` and `best-friend` roles is the inference **told-info**.

cl-exp-not-subsumes-concept

The function **cl-exp-not-subsumes-concept** is like the function **cl-exp-subsumes-concept**, except that it explains the reasons why one concept *does not subsume* (is not more general than) another concept. The syntax is

(**cl-exp-not-subsumes-concept** <derived concept> <derived concept>)

It takes two derived concepts, but it does not take a reason, origin, or all keyword. For example, consider the concept `C3` which is defined as someone with between 2 and 4 sisters and `C4`, defined to be someone with between 3 and 5 sisters. If we ask why `C4` does not subsume `C3` it will return **bad-at-least-ordering** since at least 2 does not satisfy at least 3.

cl-exp-not-subsumes-ind

The function **cl-exp-not-subsumes-ind** is like the function **cl-exp-subsumes-ind**, except that it explains the reasons why the concept *does not subsume* the individual, i.e., why the individual is not an instance of the concept. The syntax is

(**cl-exp-not-subsumes-ind** <derived concept> <derived individual>)

It takes a derived concept and a derived **CLASSIC** individual, but it does not take a reason, origin, or all keyword.

cl-exp-rule-firing

The function **cl-exp-rule-firing** takes a derived **CLASSIC** individual, and a rule that fired on the individual, and explains why the rule fired on the individual. The syntax is

(**cl-exp-rule-firing** <derived **CLASSIC** individual> <rule>
&key (reason `T`) (origin `NIL`) (all `NIL`))

The central piece of explaining why a rule fired involves explaining why the rule antecedent subsumed the individual the rule fired on. With regard to explaining that subsumption the keywords operate the same as for **cl-exp-subsumes-concept**.

As an example, suppose that a `VEGETARIAN` is defined as someone whose `food` is always of type `PLANT`: `(`**ALL** `food PLANT)`. We create a rule called `VEG-RULE` which says that if someone is a `VEGETARIAN` then he is a `HEALTHY-THING`. We then create the individual `Mary` as someone whose only `food` is of type `PLANT`. If we call **cl-exp** on `Mary`, we see that she is a `HEALTHY-THING` because the rule `VEG-RULE` fired on her. We can then call **cl-exp-rule-firing** to find out why that rule fired, i.e., how its antecedent was satisfied:

(**cl-exp-rule-firing** `@Mary` (**cl-named-rule** `'VEG-RULE`) :origin `T`)

We see that the antecedent, `VEGETARIAN`, was satisfied for the following reasons: the restriction on `VEGETARIAN` that all `food` must be of type `PLANT` is satisfied by the subsumption reason **identical-all-restr**: the individual has the identical **ALL** restriction. The origin of this information on the individual is told information.

7.1.2.1 Explanation Objects and Subsumption

Just as with the other explanation functions the functions in this section generate explanation objects which can be retrieved with **cl-get-exp**. However, the *type* of all the subsumption inferences (retrieved with **cl-exp-type**) will be **subsumption**, to distinguish these inferences from other types of explanation inferences. The arguments (retrieved with **cl-exp-args**) are as follows: the first argument for each of these inferences will be the particular type of subsumption inference, such as **at-least-ordering**, and the rest of the arguments will be the arguments to that inference. This differs from the normalization explanation inferences, where the *type* is just the inference type, e.g., **inheritance**, and the arguments are the arguments to that inference, e.g., the told parent from which the information was inherited, and the role-path along which the parent was found. Note: this only applies to inferences calculated because of the reason keyword. Those calculated because of the origin keyword are normalization inferences. The subsumption explanation functions and **cl-print-exp** can print both normalization inferences and subsumption inferences.

For the aspects **FILLS**, **PRIM**, **TEST**, **ONE-OF**, and **SAME-AS**, we store explanations (as returned by **cl-get-exp**) as a list of lists, where for each inner list, the car is the particular filler, primitive, test, **ONE-OF**, or **SAME-AS** restriction being explained, and the cdr is a list of the explanation objects that explain it. Thus, if the `child` role of the individual `Sally` has two fillers: `Sue` and `John`, and we set up explanation on the **FILLS** aspect of `Sally` for the role-path (`child`), the explanations for the **FILLS** aspect would be a list that would look something like: ((@i{`Sue`} <exp>) (@i{`John`} <exp>)). For these five aspects mentioned above, there are certain subsumption inferences which apply to all aspect-fillers for that aspect. For example, the inference **filler-subset** says that all fillers of the subsuming concept are satisfied by the fact that the subsuming concept's fillers are a subset of the subsumed object's fillers. The way we store these types of explanations for these five aspects is that instead of storing a specific filler, primitive, test, **ONE-OF**, or **SAME-AS** restriction as the car of the list, we store one of the symbols **filler-list**, **prim-list**, **test-list**, **one-of-list**, or **same-as-list**, depending on the aspect, to denote that the explanations that follow apply to all the fillers, primitives, etc., for the specified role-path on the object. Thus, for the function **cl-get-exp**, we can use one of these five symbols for the aspect-filler; or, if we leave the aspect-filler unspecified, then we might get a list of lists returned, where the car of one of the lists is one of these five symbols. These five symbols do not show up in printing, but if there are any explanations that apply to all aspect-fillers, they are printed right after the name of the aspect, and before any of the aspect-fillers are printed.

7.2 Pruning Explanation Output

A pruning mechanism has been added to explanation to make the output manageable. There are a number of variables which, when set to certain values, prune the amount of explanation information that is calculated and/or the amount of output that is printed for the explanation functions. You can set these variables individually, using **setq**; you can use the function **cl-exp-set-pruning-variables** to set these variables to certain pre-specified values; or you can just use the default settings.

The function **cl-exp-set-pruning-variables** takes an optional argument, and sets the pruning variables as follows:

- if the argument is unspecified, it sets the pruning variables to their default values;
- if the argument is `T`, it turns on all pruning (this is very close to the default setting);
- if the argument is `NIL`, it turns off all pruning (prints verbosely).

The pruning variables are as follows:

- ***cl-exp-print-length*** (default is `3`). Sometimes during explanation printing, the expression for a concept is printed. If this variable is non-`NIL` (it must be an integer of length at least `1`), then it acts like the Common LISP variable ***print-length***, and expressions will only be printed up to this length, and then "..." will be printed.
- ***cl-exp-print-level*** (default is `3`). Sometimes during explanation printing, the expression for a concept is printed. If this variable is non-`NIL` (it must be an integer of length at least `1`), then it acts like the Common LISP variable ***print-level***, and expressions will only be printed up to this level, and then # will be printed.
- ***cl-exp-ignore-at-least*** (default is `NIL`). If the aspect is `T` and this variable is `T`, don't explain **AT-LEAST** restrictions.
- ***cl-exp-ignore-at-most*** (default is `NIL`). If the aspect is `T` and this variable is `T`, don't explain **AT-MOST** restrictions.

- ***cl-exp-ignore-propagation-filler*** (default is T). For complete explanation, if this variable is T, don't explain the filler of a propagation, just the information propagated (e.g., if Mary is an ATHLETE because Sue's child is Mary and Sue contains the restriction (**ALL** child ATHLETE), then if this variable is T, explain how Sue got the **ALL** restriction, but don't explain how Sue got child Mary).

- ***cl-exp-ignore-top-prims*** (default is T). If this variable is T, don't explain CLASSIC-THING or HOST-THING.

- ***cl-exp-ignore-all-prims*** (default is NIL). If this variable is T, don't explain any primitives.

- ***cl-exp-ignore-intervals*** (default is T). For complete explanation, if this variable is T, don't explain **interval-length-equals-at-least-implies-fillers** (i.e., if fillers got added because of an interval, don't explain where the **MIN** and **MAX** restrictions came from).

- ***cl-exp-ignore-interval-tests*** (default is T). If a concept contains an interval (**MIN** and/or **MAX** restriction), don't explain the tests **atom**, **numberp**, **rationalp**, or **integerp**.

- ***cl-exp-ignore-redundant-propagations*** (default is T). Only explain the most specific propagations. For example, if there is a role hierarchy, and propagation p1 is along an identical or more specific role-path than propagation p2, and p1 propagates an identical or more specific restriction than p2, just explain p1, and not p2.

- ***cl-exp-ignore-redundant-one-ofs*** (default is T). Only explain the final **ONE-OF** restriction, and possibly the most specific **ONE-OF** restrictions, whose intersection generated the final **ONE-OF** restriction.

- ***cl-exp-tests-level*** (default is 0). Governs the amount of information to present about test subsumption during explanation. Its values can be 0, 1 or 2. A value of 0 means that CLASSIC should indicate only that the subsumer's tests subsume the subsumee's tests in some way. A value of 1 means to show explicitly one subsumed test for each test in the subsuming concept. A value of 2 means to show explicitly every subsumed test for each test in the subsuming concept. For more details, see Section 9.6.1.

- ***cl-complete-exp-ignore-rules*** (default is NIL). In complete explanation on an individual, when a piece of information is explained by a rule firing, the next step is to explain why the rule fired, i.e., why the antecedent of the rule was satisfied. If this pruning variable is T, the rule firing will not be explained.

These pruning variables are included for backwards compatibility. In knowledge base development, we suggest using meta information (see Section 8) to handle the work done by the aspect-oriented pruning variables – ***cl-exp-ignore-at-least***, ***cl-exp-ignore-at-most***, ***cl-exp-ignore-top-prims***, ***cl-exp-ignore-all-prims***, ***cl-exp-ignore-intervals***, ***cl-exp-ignore-interval-tests***, and ***cl-exp-ignore-redundant-one-ofs***.

7.3 Using Explanation to Understand Errors

Explanation can be used to help explain error conditions and incoherencies. As discussed in Section 6, when a "real" error is encountered as the result of a CLASSIC function call, CLASSIC returns the symbol `classic-error` as the first return value, the error type and arguments as the second value, the error role-path as the third value, and a CLASSIC concept or individual that had the error on it as the fourth value. In order to use explanation on an error-producing statement, the user can bind the fourth value and then use explanation functions on it.

For a simple example, suppose that the concepts `HEALTHY-THING` and `UNHEALTHY-THING` are primitively disjoint. If we define the concept `C1` as:

(**cl-define-concept** `'C1 '(`**AND** `HEALTHY-THING UNHEALTHY-THING))`

we get the error DISJOINT-PRIMS-CONFLICT. To explain the error, we can bind the values that **cl-define-concept** returns:

```
(multiple-value-setq (concept err role-path bad-concept)
    (cl-define-concept 'C1 '(AND HEALTHY-THING UNHEALTHY-THING)))
```

Then a simple call to the explanation function on the bad-concept:

(**cl-exp** `bad-concept)`

will produce the desired explanation. In this case it would explain that a DISJOINT-PRIMS-CONFLICT occurred between the concepts `HEALTHY-THING` and `UNHEALTHY-THING`, and each of these concepts came from told information. Note: a simpler way of getting this explanation without the **multiple-value-setq** would be to use the **cl-error-object** function described below. However, by using **multiple-value-setq**, a program can automatically trap an error case. To get this same explanation without the **multiple-value-setq**, you can call:

(**cl-exp** (**cl-error-object** `'C1`))

after getting the error (see below).

Because certain explanation information is pruned out by default, you may need to turn pruning off to get an explanation for certain errors. The **AT-LEAST**, **AT-MOST**, **MIN** and **MAX** aspects are pruned by default. Thus, if you generate a conflict between the **AT-LEAST** and **AT-MOST** aspects, you will need to turn pruning off to get an explanation of the error. For example, the following definition of `C2` produces an error:

```
(cl-define-concept 'C2
   '(AND (AT-LEAST 2 r2) (AT-MOST 0 r2)))
```

In order to explain the error, we must bind the values that **cl-define-concept** returns, and then turn off pruning to get the explanation:

```
(multiple-value-setq (concept2 err2 role-path2 bad-concept2)
    (cl-define-concept 'C2
       '(AND (AT-LEAST 2 r2) (AT-MOST 0 r2))))
```

Now you can either turn off all pruning, with

(**cl-exp-set-pruning-variables** `NIL`)

followed by a call to

(**cl-exp** bad-concept2)
or you can temporarily turn off **AT-LEAST** and **AT-MOST** pruning while doing an explanation as follows:

(**let** ((***cl-exp-ignore-at-least*** `NIL`)
(***cl-exp-ignore-at-most*** `NIL`))
(**cl-exp** bad-concept2))

The explanation will tell you that an inconsistent bounds conflict occurred on the role `r2`, the bounds were **AT-LEAST** `2` and **AT-MOST** `0`, and they each came from told information. Note: you can always try explaining an error with pruning turned on, and then if it doesn't give enough of an explanation, you can run explanation again with pruning off. The simplest way is to just turn off all pruning with (**cl-exp-set-pruning-variables** `NIL`)—that way you don't have to worry about which aspect of pruning to turn off. Then afterwards you can turn pruning back on with (**cl-exp-set-pruning-variables**) or (**cl-exp-set-pruning-variables** `T`), depending on whether you want the default pruning or complete pruning, respectively.

Sometimes, just having access to the single error object on which the error occurred is not good enough. In some cases, adding information to one individual, `I1`, can cause a conflict on a second individual, `I2`. When you ask for an explanation on the error individual for `I2`, CLASSIC will say that the information was propagated from `I1`. If you then ask for an explanation on `I1`, you will not get the desired result, because the information that was added to `I1` that caused the conflict on `I2`, was automatically retracted from `I1` to get the knowledge base back into a consistent state. Therefore, when a "real" error is encountered in CLASSIC, copies of all individuals which were modified as a result of the problematic input are stored on the list ***cl-error-object-list***. Each individual on this list contains information derived from any inferences performed based on the problematic input. The individuals on this list can be used to trace the source of each piece of information which caused an error.

Suppose that the concepts `MALE` and `FEMALE` are disjoint, the concept `MALE-CHILDREN` is someone all of whose children are `MALE`, and the concept `FEMALE-CHILDREN` is someone all of whose children are `FEMALE`. Now suppose that `Sue` is a `FEMALE`, and that `Mary` is created (perhaps mistakenly) under the concept `MALE-CHILDREN`. What happens when we try to add the information that `Mary`'s `child` is `Sue`?

```
(cl-startup)
(cl-define-primitive-role 'child)
(cl-define-disjoint-primitive-concept 'MALE 'CLASSIC-THING 'gender)
(cl-define-disjoint-primitive-concept 'FEMALE 'CLASSIC-THING 'gender)
(cl-define-concept 'MALE-CHILDREN '(ALL child MALE))
(cl-define-concept 'FEMALE-CHILDREN '(ALL child FEMALE))
(cl-create-ind 'Sue 'FEMALE)
(cl-create-ind 'Mary 'MALE-CHILDREN)
(cl-ind-add @Mary '(FILLS child Sue))
```

We get the following error:

```
*CLASSIC ERROR* while processing
  (cl-ind-add @i{Mary} (FILLS child Sue))
    occurred on object @i{Sue-*Incoherent*}:
  Trying to combine disjoint primitives: @tc{FEMALE} and @tc{MALE}.
```

We get (**disjoint-prims-conflict** @tc{`FEMALE`} @tc{`MALE`}) returned as the error code and arguments (the second return value, which we can access by using a **multiple-value-setq**). Now we can ask where the information on the error object for `Sue` came from:

(**cl-exp** (**cl-error-object** 'Sue))

The system will show us that the primitive concept `MALE` was propagated from individual `Mary` and role `child`, and the primitive concept `FEMALE` was told information.

Then we can ask where the information on the error object for `Mary` came from:

(**cl-exp** (**cl-error-object** 'Mary))

The system will show us that for the `child` role, the **ALL** restriction that children must be `MALE` was inherited from the concept `MALE-CHILDREN`.

At this point, we would realize that we had entered the wrong information about `Mary`—we meant to say that she was under the concept `FEMALE-CHILDREN`. We could then fix the situation, and add `Sue` as a `child`:

```
(cl-ind-remove-concepts @Mary (list @tc{MALE-CHILDREN}))
(cl-ind-add-concepts @Mary (list @tc{FEMALE-CHILDREN}))
(cl-ind-add @mary '(FILLS child Sue))
```

Then if we call **cl-exp** on `Mary` we see that the new **ALL** restriction that children must be `FEMALE` gets inherited from the concept `FEMALE-CHILDREN`. See Section 6 for a complete description of error handling in CLASSIC.

In addition to calling **cl-exp** on errors to explain either general information on an error object, or a specific role-path, aspect, and aspect-filler on an error object, there is a specialized function we can use for explaining errors: **cl-exp-error**. Given an error object, this function calculates an explanation for exactly the information which led to the error, based on the type of error. This function does not take as arguments a role-path, aspect, and aspect-filler, but instead, calculates the appropriate pieces of explanation for each type of error. For example, for the error **inconsistent-bounds-conflict**, it would explain where the conflicting **AT-LEAST** and **AT-MOST** restrictions came from. This function only does a one-step explanation, like that done by **cl-exp**, not a complete explanation, like that done by **cl-complete-exp**. The advantage is that it explains exactly the information leading to the error. If there is lots of extraneous information in the object that is unrelated to the error, this information will not get explained, like it would in a general call to **cl-exp**. Also, when **cl-exp-error** is called, it is not necessary to adjust the pruning variables to get the desired information, because this information will not get pruned out.

In the example for concept **C2** above, when **cl-exp** was called, some pruning variables had to be turned off first. Suppose we try to define **C2** as before:

```
(cl-define-concept 'C2
  '(AND (AT-LEAST 2 r2) (AT-MOST 0 r2)))
```

If we call

```
(cl-exp-error (cl-error-object 'C2))
```

we get the following output, which is the same that we would get if we called **cl-exp** with the pruning variables turned off:

```
C2-*INCOHERENT* -> :  (INCONSISTENT-BOUNDS-CONFLICT @r{R2} 2 0)
 Role Restrictions:
  R2
   at-least:  (2) :  TOLD-INFO
   at-most:  (0) :  TOLD-INFO
```

7.4 Explanation Inferences

CLASSIC performs many different types of inferences. The ones which CLASSIC explains are described below. Note that the explanation objects that are described in Section 7.1 (returned by **cl-get-exp**, and used by **cl-exp-type** and **cl-exp-args**) contain both the type of the inference, and the arguments associated with it. This information is displayed by the printing functions (**cl-exp**, **cl-print-exp**, **cl-exp-subsumes-concept**, **cl-exp-subsumes-ind**, and **cl-exp-rule-firing**).

There are two different kinds of inferences: normalization inferences and subsumption inferences. The subsumption inferences are those which explain why one object subsumes another object. They are generated by the subsumption explanation functions: **cl-exp-subsumes-concept**, **cl-exp-subsumes-ind**, and **cl-exp-rule-firing**). For the normalization inferences, the inference types (retrieved with **cl-exp-type**) are the types listed below. However, for the subsumption inferences, the type is always **subsumption**, to distinguish the subsumption inferences from the normalization inferences. For the subsumption inferences, the first argument (from the list returned by **cl-exp-args**) is the particular type of subsumption inference, and the rest of the arguments are the arguments to that inference.

As a general rule, the inferences which contain the word **implies** are normalization inferences, and those which contain the word **satisfies** are subsumption inferences.

In the following descriptions of the inferences, the arguments to each inference are listed after the inference type, enclosed within [ ]'s. [ ] means no arguments.

7.4.1 Normalization Inferences

at-least-and-one-of-implies-fillers [ ] When the **AT-LEAST** restriction on a role equals the length of the **ONE-OF** restriction, e.g.,
(**AND** (**AT-LEAST** 2 `grape`) (**ALL** `grape` (**ONE-OF** `Zinfandel Petite-Syrah`)))
then the fillers for the role must be exactly the individuals in the **ONE-OF** restriction (in this case, the `grape` role must be filled by `Zinfandel` and `Petite-Syrah`).

attribute-implies-at-most-1 [ ] If a role is defined to be an attribute, then the **AT-MOST** restriction is set to `1`, because attributes can have at most `1` filler.

classic-ind-implies-classic-thing [ ] A top-level **CLASSIC** individual will have the primitive `CLASSIC-THING` in its primitive list (because it must be in the **CLASSIC** realm). This inference is calculated during explanation only if `CLASSIC-THING` is not there for any other reason. For the functions **cl-exp** and **cl-print-exp**, this inference is not shown unless the all keyword is non-`NIL`.

closed-implies-at-most [ ] If a role on an individual is closed, then the maximum number of fillers the role can have is the current number of fillers. The **AT-MOST** restriction is set to this number.

fillers-implies-at-least [ ] If a role has known fillers (for example, the `grape` role might be filled with `Shiraz` and `Cabernet-Sauvignon`), then the **AT-LEAST** restriction on the role is set to the number of fillers (in this example, 2). (If there was already a more specific **AT-LEAST** restriction, this inference would not be applied.)

filter-one-of-by-interval [ ] Similar to **filter-one-of-by-tests**, this inference removes values in a **ONE-OF** restriction that are not in the specified interval.

filter-one-of-by-tests [ ] Values in **ONE-OF** restrictions are filtered by tests. If for example something is known to be an `INTEGER` (and thus has the **integerp** test) and has a **ONE-OF** restriction of {1 1.2 5}, then this inference will remove the 1.2 value.

inconsistent-all-implies-at-most-zero [ ] If an inconsistent **ALL** restriction is encountered, such as (**ALL** `child` (**AND** `FEMALE MALE`)) then the deduction is made that there can be no `children`, and the **AT-MOST** restriction is set to `0`.

inheritance [(1) a told concept from which the information was inherited; (2) the role-path along which the concept was found] For object `A`, if `A` is a subclass or instance of `B`, and `B` has a property C, then `A` inherits property C from `B`.

interval-and-integer-implies-at-most [(1) the **MIN** restriction; (2) the **MAX** restriction] When a role is restricted to type `INTEGER`, and the complete interval is specified (both the **MIN** and the **MAX**), the **AT-MOST** restriction is set to the number of integers in the interval. For example, if an object has the restriction

(**ALL** `age` (**AND** `INTEGER` (**MIN** `15`) (**MAX** `18`)))

then the **AT-MOST** restriction is set to `4`.

interval-implies-at-most [(1) the **MIN** restriction; (2) the **MAX** restriction] If a role has an interval, where both the `MIN` and `MAX` are specified and are equal, then there can be at most one filler for the role, and the **AT-MOST** restriction is set to `1`.

interval-implies-host-thing [ ] If a concept or individual has a **MIN** or **MAX** restriction, then it will have the primitive `HOST-THING` added to its primitive list (because it must be in the **HOST** realm). This inference is calculated during explanation only if `HOST-THING` is not there for any other reason. For the functions **cl-exp** and **cl-print-exp**, this inference is not shown unless the all keyword is non-`NIL`.

interval-length-equals-at-least-implies-fillers [ ] Similar to **at-least-and-one-of-implies-fillers**, when the interval length equals the **AT-LEAST** restriction, and the role is restricted to type `INTEGER`, then the fillers must be exactly the integers in the interval. For example, if an object has the restriction

(**AND** (**AT-LEAST** 3 `favorite-number`)
    (**ALL** `favorite-number` (**AND** `INTEGER` (**MIN** `14`) (**MAX** `16`))))

then the fillers for `favorite-number` get set to `14`, `15`, and `16`.

interval-test [ ] If we know that something is in an interval, then we know that it must be a NUMBER. Thus, if test restrictions containing the functions **numberp** and **atom** are not already on this object, then this inference adds them.

inverse-propagation [(1) the role on the object whose **ALL** restriction led to this inference; (2) the role-path along which the **ALL** restriction was found; (3) the information which was propagated—can be either a named or an unnamed concept. Note: if the information which was propagated is identical to the information currently being explained, the explanation printing functions will not print this argument, but it will still be returned from **cl-exp-args**; (4) if the **ALL** restriction on the first role is a named concept, which contains a role restriction on the inverse role, which is what got propagated, then this argument is the named concept. Otherwise, it is NIL.] If role inverses r1 and r2 are defined, and individual I1 has the following restriction (either at the top level, or on an **ALL** restriction):

(**AND** (**AT-LEAST** 1 r1) (**ALL** r1 (**ALL** r2 **C**)))

for some description **C**, then **C** gets propagated onto I1 (or onto the **ALL** restriction, if it was not at the top level). For example, if we know that child and parent are inverse roles, and we know that a particular individual, Deb, has at-least 1 parent and all of her parents have only ATHLETEs as children, then Deb must be an ATHLETE. The argument in this example is the parent role, indicating that all of Deb's parent's children are ATHLETEs.

inverse-role [(1) the individual whose role was known to be restricted (before this inference ran); (2) the role on that individual] If role inverses r1 and r2 are defined, then if individual I1 has role r1 filled with individual I2, then I2's r2 role is automatically filled with I1. If, for example, parent and child are inverse roles, and CLASSIC is told that Mary's child is Bob, then Bob's parent role gets filled with Mary. The arguments in this example are Mary and child, indicating that Mary's child is Bob.

one-of-implies-at-most [ ] If an **ALL** restriction on a role contains a **ONE-OF** restriction, then the largest number of fillers the role can have is the number of individuals in the **ONE-OF** restriction, and the **AT-MOST** restriction is set to this number. For the description

(**ALL** color (**ONE-OF** White Red Rose))

the **AT-MOST** restriction on the color role is set to 3.

one-of-implies-classic-thing [ ] If a concept or individual has a **ONE-OF** restriction containing **CLASSIC** individuals, then it will have the primitive CLASSIC-THING added to its primitive list (because it must be in the **CLASSIC** realm). This inference is calculated during explanation only if CLASSIC-THING is not there for any other reason. For the functions **cl-exp** and **cl-print-exp**, this inference is not shown unless the all keyword is non-NIL.

one-of-implies-host-thing [ ] If a concept or individual has a **ONE-OF** restriction containing **HOST** individuals, then it will have the primitive HOST-THING added to its primitive list (because it must be in the **HOST** realm). This inference is calculated during explanation only if HOST-THING is not there for any other reason. For the functions **cl-exp** and **cl-print-exp**, this inference is not shown unless the all keyword is non-NIL.

one-of-intersection [ ] Different **ONE-OF** restrictions have been intersected to form the final **ONE-OF** restriction. For example, if a concept has two **ONE-OF** restrictions: (**ONE-OF** Mary Jane Jack) and (**ONE-OF** Jack Jill Susan), either through told-info, inheritance, or some other inference, then the final **ONE-OF** restriction will be (**ONE-OF** Jack Jill).

propagation [(1) the individual from which the information was propagated; (2) either a role or list of roles—the role or role-path through which the information was propagated; (3) the information which was propagated—can be either a named or an unnamed concept.] If information about one individual needs to be transferred to another individual because one individual fills a role of another individual, then this information is said to be propagated. If, for example, we know that PARENT-OF-ATHLETES describes individuals which have only ATHLETEs as children, and we know that Joe is a PARENT-OF-ATHLETES, and we know that Joe's child is Tom, then we also know that Tom must be an ATHLETE. The information that Tom is an ATHLETE is propagated from the individual Joe through the child role.

It is also possible that information can be propagated from one individual to another through **ALL** restrictions and a **FILLS** restriction. For example, suppose that we don't know who Jack's friends are, but we know that he has at least 1 friend, all his friends have Susan as a teacher, and all his friends' teachers are VEGETARIANS. Then we know that Susan must be a VEGETARIAN. This information is propagated from the individual Jack through the friend teacher role-path.

role-hierarchy-inherit-all [(1) the parent role from which the information was inherited; (2) the role-path along which the **ALL** restriction was found, not including the final parent role. (The second argument may be needed if the **ALL** restriction which is inherited is not a named concept.)] If there is an **ALL** restriction on a role, then all child roles inherit this **ALL** restriction. For example, if an object has a wine role with a value restriction that restricts all fillers to be instances of the concept WINE, then the subroles white-wine and red-wine would inherit this restriction from the wine role. The argument in this example is the wine role.

role-hierarchy-inherit-at-most [(1) the parent role from which the information was inherited] If a role hierarchy exists, and there is an **AT-MOST** restriction on a role, then its child roles inherit the **AT-MOST** restriction. If, for example, an object has a wine role with an **AT-MOST** restriction of 3 and there are two subroles of wine, white-wine and red-wine, then both white-wine and red-wine would inherit the **AT-MOST** restriction of 3 from their parent role wine.

role-hierarchy-inherit-at-least [(1) the child role from which the information was inherited] If a role hierarchy exists, and there is an **AT-LEAST** restriction on one of the child roles, then the **AT-LEAST** restriction must be "inherited"[16] up the hierarchy to the parent role and all ancestor roles. If in the above example, the `white-wine` role has an **AT-LEAST** restriction of 2, then the parent role `wine` also gets an **AT-LEAST** restriction of 2. The argument in this example is the `white-wine` role.

role-hierarchy-inherit-fills [(1) the child role from which the information was inherited] If a role hierarchy exists, and fillers are known for a subrole, then the fillers must be "inherited" up the hierarchy to all ancestor roles. If in the above example, the `white-wine` role had a filler of `Forman-Chardonnay`, then the parent role `wine` would also get the filler `Forman-Chardonnay`. The argument in this example is the `white-wine` role.

role-restr-implies-classic-thing [ ] If a concept or individual has a role restriction (**FILLS**, **ALL**, **AT-LEAST** or **AT-MOST**), then it will have the primitive `CLASSIC-THING` added to its primitive list (because it must be in the **CLASSIC** realm). This inference is calculated during explanation only if `CLASSIC-THING` is not there for any other reason. For the functions **cl-exp** and **cl-print-exp**, this inference is not shown unless the all keyword is non-`NIL`.

rule [the list of rules which generated this information] If a rule firing caused something to be derived, then the rule will be the explanation for that fact. For example, if the rule `VEG-RULE` says that all `VEGETARIAN`s must be `HEALTHY-THING`s, and `Mary` is classified as a `VEGETARIAN`, then the fact that she is a `HEALTHY-THING` is explained by the rule `VEG-RULE`.[17]

same-as [(1) and (2) the two role paths from the **SAME-AS** restriction; (3) the role-path along which the **SAME-AS** restriction was found] If a **SAME-AS** restriction exists and one role path has an **ALL** restriction which is a description (not a named concept), then every piece of that description gets propagated to the other role path in the **SAME-AS** restriction. For example, if `Mary`'s `best-friend` is the same as her `sister`, and her `best-friend` has at least 1 `pet`, then her `sister` has at least 1 `pet`, by the **same-as** inference.

same-as-implies-all [(1) and (2) the two role paths from the **SAME-AS** restriction; (3) the role-path along which the **SAME-AS** restriction was found] If a **SAME-AS** restriction exists and one role path has an **ALL** restriction which is a named concept, then the other role path in the **SAME-AS** restriction will have the same **ALL** restriction.

same-as-implies-at-least-1 [(1) and (2) the two role paths from the **SAME-AS** restriction; (3) the role-path along which the **SAME-AS** restriction was found] When a **SAME-AS** restriction is asserted on an object, all the attributes in both paths of the **SAME-AS** restriction get **AT-LEAST** restrictions of `1`.

[16] Typically, one thinks of information as being inherited down a hierarchy, but we also use the word "inherit" when information moves up a hierarchy.

[17] We do not guarantee a complete explanation if one rule firing causes another rule to fire which in turn causes the atomic fact being explained to become more specific.

same-as-implies-classic-thing [ ] If a concept or individual has a **SAME-AS** restriction, then it will have the primitive `CLASSIC-THING` added to its primitive list (because it must be in the **CLASSIC** realm). This inference is calculated during explanation only if `CLASSIC-THING` is not there for any other reason. For the functions **cl-exp** and **cl-print-exp**, this inference is not shown unless the all keyword is non-`NIL`.

same-as-implies-filler [(1) and (2) the two role paths from the **SAME-AS** restriction; (3) the role-path along which the **SAME-AS** restriction was found] If a **SAME-AS** restriction exists and one role path has a known filler, the other role path in the **SAME-AS** restriction will have the same filler.

subrole-combination-implies-at-least [the subroles used to calculate the restriction] When a role hierarchy exists, subrole combinations can come into play. If two child roles have disjoint **ALL** restrictions (i.e., they must be filled with distinct individuals), then we can calculate a minimum for the **AT-LEAST** restriction on the parent role by just adding up the **AT-LEAST** restrictions on the child roles. For example, if `relative, brother`, and `sister` are all roles and we know that `sister` must be `FEMALE`, `brother` must be `MALE`, and there is at least `1 sister` and at least `1 brother`, then we know that there is an **AT-LEAST** restriction of `2` on `relative`.

This inference also applies when two child roles have **ALL** restrictions that are not disjoint, but have a finite overlap. This would occur when at least one of the roles has a **ONE-OF** restriction. In this case, the maximum number of instances they could have in common is calculated, and a new **AT-LEAST** restriction is calculated based on this.

subrole-combination-implies-at-most [(1) the parent role that has the **AT-MOST** restriction; (2) a list of the subroles that have the **AT-LEAST** restrictions] Given a role hierarchy, an **AT-MOST** restriction on a parent role can combine with **AT-LEAST** restrictions on child roles to produce an **AT-MOST** restriction on a different child role. Given the roles `relative, brother`, and `sister` defined above, where the **ALL** restrictions on `brother` and `sister` are disjoint, then if we have an **AT-MOST** `5 relative` restriction, and we know there are **AT-LEAST** `2 brother`s, then this inference is used to deduce that there are **AT-MOST** `3 sister`s.

test-c-implies-classic-thing [ ] If a concept or individual has a **TEST-C** restriction, then it will have the primitive `CLASSIC-THING` added to its primitive list (because it must be in the **CLASSIC** realm). This inference is calculated during explanation only if `CLASSIC-THING` is not there for any other reason. For the functions **cl-exp** and **cl-print-exp**, this inference is not shown unless the all keyword is non-`NIL`.

test-h-implies-host-thing [ ] If a concept or individual has a **TEST-H** restriction, then it will have the primitive `HOST-THING` added to its primitive list (because it must be in the **HOST** realm). This inference is calculated during explanation only if `HOST-THING` is not there for any other reason. For the functions **cl-exp** and **cl-print-exp**, this inference is not shown unless the all keyword is non-`NIL`.

told-info [ ] CLASSIC was directly told this information.

7.4.2 Subsumption Inferences

Note that the following inferences all have **subsumption** as the inference type, and the particular type of subsumption inference (from the inferences below) is stored as the first argument (retrieved by **cl-exp-args**). This is only important to know if you use the functions **cl-get-exp**, **cl-exp-type**, and **cl-exp-args**. The high-level subsumption explanation functions (**cl-exp-subsumes-concept**, **cl-exp-subsumes-ind**, and **cl-exp-rule-firing**), and the printing function **cl-print-exp** should make this transparent. When they display the calculated explanations, they will print **Subsumption:** before the subsumption inferences, and **Origin:** before the normalization inferences, unless there are no subsumption inferences, in which case neither **Subsumption:** nor **Origin:** will get printed.

For the following inferences, if the name starts with **ind**, the subsumee must be an individual. For the rest of the inferences, the piece of the subsumee being explained can be either a concept (or an **ALL** restriction) or an individual, unless the documentation states otherwise.

at-least-ordering [(1) subsumee **AT-LEAST**; (2) subsumer **AT-LEAST**] The **AT-LEAST** restriction on the subsumee is no smaller than the **AT-LEAST** on the subsumer. For example, `(`**AT-LEAST** `3 r)` is subsumed by `(`**AT-LEAST** `2 r)`.

at-most-ordering [(1) subsumee **AT-MOST**; (2) subsumer **AT-MOST**] The **AT-MOST** restriction on the subsumee is no larger than the **AT-MOST** on the subsumer.

bad-at-least-ordering [(1) "subsumee" **AT-LEAST**; (2) "subsumer" **AT-LEAST**] The **AT-LEAST** restriction on the "subsumee" is smaller than the **AT-LEAST** on the "subsumer".

bad-at-most-ordering [(1) "subsumee" **AT-MOST**; (2) "subsumer" **AT-MOST**] The **AT-MOST** restriction on the "subsumee" is larger than the **AT-MOST** on the "subsumer".

bad-max-ordering [(1) "subsumee" **MAX**; (2) "subsumer" **MAX**] The **MAX** restriction on the "subsumee" is larger than the **MAX** on the "subsumer".

bad-min-ordering [(1) "subsumee" **MIN**; (2) "subsumer" **MIN**] The **MIN** restriction on the "subsumee" is smaller than the **MIN** on the "subsumer".

doesnt-satisfy-incoherent [(1) the incoherent **ALL** restriction on the "subsumer"; (2) the **ALL** restriction on the "subsumee"] An incoherent **ALL** restriction cannot be satisfied.

filler-subset [ ] The fillers of the subsumer are a subset of the fillers of the subsumee.

fillers-and-at-most-satisfies-all [(1) subsumee fillers; (2) subsumer **ALL** restriction] If all the fillers of a role `r1` on an individual `I1` are known (the number of fillers equals the **AT-MOST** restriction), and each of the fillers satisfies a description `D1`, then `I1` satisfies the description `(`**ALL** `r1 D1)`. For example, if `Mary` and `Bill` are both known to be `ATHLETES`, and `Sue` is defined as

`(`**AND** `(`**FILLS** `child Mary Bill) (`**AT-MOST** `2 child))`, then `Sue` satisfies the description

`(ALL child ATHLETE)`. This inference also applies to concept subsumption, when the subsuming **ALL** restriction is a **HOST** concept, and the subsumed concept's fillers are **HOST** individuals.

fillers-dont-satisfy-all [(1) "subsumee" fillers which don't satisfy the **ALL** restriction; (2) "subsumer" **ALL** restriction] Not all the fillers satisfy the **ALL** restriction.

fillers-missing [(1) the fillers not satisfied] The fillers of the "subsumer" are not a subset of the fillers of the "subsumee".

identical-all-restr [no arguments, if both **ALL** restrictions are the same classified concept; otherwise: (1) subsumer **ALL** restriction; (2) subsumee **ALL** restriction] The **ALL** restriction on the subsumer is equivalent to the **ALL** restriction on the subsumee.

ind-doesnt-pass-tests [(1) the tests the individual doesn't pass] The individual doesn't satisfy a test restriction. An individual can satisfy a test if running the test on the individual returns `T`, or if the individual has the identical test in its descriptor.

ind-doesnt-satisfy-same-as [(1) the **SAME-AS** restrictions the individual doesn't satisfy] The individual doesn't satisfy a **SAME-AS** restriction.

ind-member-of-one-of [ ] The individual satisfies a **ONE-OF** restriction because it is one of the individuals in the **ONE-OF**. There is no other way that a top-level individual can satisfy a **ONE-OF** restriction.

ind-not-member-of-one-of [(1) the **ONE-OF** restriction] The individual does not satisfy the **ONE-OF** restriction because it is not one of the individuals in the **ONE-OF**.

ind-passes-test [ ] The individual satisfies a test restriction because when the test runs on the individual, it returns `T`. Note: an individual can also satisfy a test if it has the identical test in its descriptor (**test-existence**). The test gets printed during explanation.

ind-satisfies-interval [(1) the piece of the interval currently being explained (the **MIN** or the **MAX**)] For a top-level **HOST** individual: for the **MIN** and/or **MAX**, whichever are specified, the individual is no smaller than the **MIN**, and no larger than the **MAX**.

ind-satisfies-same-as-with-filler-and-paths [(1) the intermediate individual, (2) and (3) the two paths on the individual which are equivalent, and (4) optionally, a list containing pairs of a **SAME-AS** restriction which contributes to those paths being equivalent, along with the role-path along which the **SAME-AS** was found. The final argument is not there if the two paths are the two paths of a **SAME-AS**] The individual satisfies the **SAME-AS** because there is an intermediate individual along both the paths of the **SAME-AS**, and then the rest of both paths are equivalent somehow through other **SAME-AS** restrictions. For example, suppose concept `C1` is defined as: `(SAMEAS (favorite-teacher best-friend) (mother husband))`. Now suppose that `Jane`'s `favorite-teacher` and `mother` are both `Sally`, and `Sally` has the restriction: **(SAME-AS** `(best-friend) (husband))`. If we call **(cl-exp-subsumes-ind** `@C1 @Jane)`, we see that the **SAME-AS** restriction is satisfied by **ind-satisfies-same-as-with-filler-and-paths**, with the intermediate individual `Sally`, and the equivalent tail paths: `(husband)` and `(best-friend)`. Note: this inference only applies to individuals; see the **SAME-AS** example towards the end of section 4.1, for an explanation.

interval-satisfies-one-of [(1) subsumee **MIN**; (2) subsumee **MAX**; (3) subsumer **ONE-OF**] The subsumee is restricted to being an `INTEGER`. The **MIN** and **MAX** are both specified, and each integer from the **MIN** through the **MAX** is in the subsumer's **ONE-OF** restriction. This inference will never appear on a top-level individual.

max-ordering [(1) subsumee **MAX**; (2) subsumer **MAX**] The **MAX** restriction on the subsumee is no larger than the **MAX** on the subsumer. This inference will never appear on a top-level individual.

min-ordering [(1) subsumee **MIN**; (2) subsumer **MIN**] The **MIN** restriction on the subsumee is no smaller than the **MIN** on the subsumer. This inference will never appear on a top-level individual.

non-subsuming-all-restr [(1) the "subsumer" **ALL** restr; (2) the "subsumee" **ALL** restr] The **ALL** restriction on the "subsumee" doesn't satisfy the **ALL** restriction on the "subsumer".

one-of-not-satisfied [(1) "subsumee" **ONE-OF** restriction; (2) "subsumer" **ONE-OF** restriction] The **ONE-OF** in the "subsumee" contains individuals not in the **ONE-OF** of the "subsumer".

one-of-satisfies-max [(1) subsumee **ONE-OF** restriction; (2) subsumer **MAX** restriction] All the individuals in the **ONE-OF** restriction are numbers, and each one is no larger than the **MAX**. This inference will never appear on a top-level individual.

one-of-satisfies-min [(1) subsumee **ONE-OF** restriction; (2) subsumer **MIN** restriction] All the individuals in the **ONE-OF** restriction are numbers, and each one is no smaller than the **MIN**. This inference will never appear on a top-level individual.

one-of-satisfies-tests [(1) the subsumee's **ONE-OF**] Every individual in the subsumee's **ONE-OF** restriction satisfies all the subsumer's tests. This inference will never appear on a top-level individual.

one-of-subset [ ] The individuals in the **ONE-OF** restriction on the subsumee are a subset of the individuals in the **ONE-OF** restriction on the subsumer. This inference will never appear on a top-level individual.

primitive-subset [ ] The primitives of the subsumer are a subset of the primitives of the subsumee.

prims-not-satisfied [(1) the primitives not satisfied] The primitives of the "subsumer" are not a subset of the primitives of the "subsumee".

role-not-closed [(1) the "subsumer" **ALL** restriction; (2) the role] The fillers cannot satisfy the **ALL** restriction, because the role is not closed on the individual.

same-as-existence [ ] The subsumee satisfies a **SAME-AS** restriction because it has the identical **SAME-AS** restriction.

same-as-not-satisfied [(1) the **SAME-AS** restrictions not satisfied] Not all the **SAME-AS** restrictions are satisfied.

same-as-subset [ ] The **SAME-AS** restrictions of the subsumer are a subset of the **SAME-AS** restrictions of the subsumee.

satisfies-same-as-with-filler [(1) the filler] The same filler is at the end of each of the two paths of the **SAME-AS** restriction.

subsuming-all-restr [(1) subsumer **ALL** restriction; (2) subsumee **ALL** restriction] The **ALL** restriction on the subsumer subsumes the **ALL** restriction on the subsumee. If you want to find out why, you can call **cl-exp-subsumes-concept** on the two **ALL** restrictions.

test-does-not-subsume [(1) subsumer test; (2) subsumee test] The subsumer test restriction does not subsume the subsumee test restriction. There is another version of this inference (see immediately below).

test-does-not-subsume [(1) user-generated explanation, as a list] One test did not subsume another for the reason given by the argument, a programmer-generated explanation. See Section 9 for more details. There is another version of this inference (see immediately above).

test-existence [ ] The subsumee satisfies a test restriction because it has the identical test restriction. The test gets printed during explanation.

test-subset [ ] The test restrictions of the subsumer are a subset of the test restrictions of the subsumee by virtue of textual equivalence.

test-subset-by-subsumption [ ] Each test restriction of the subsumer subsumes at least one of the test restrictions of the subsumee.

test-subsumes [(1) subsumer test; (2) subsumee test] The subsumer test restriction subsumes the subsumee test restriction. There is another version of this inference (see immediately below).

test-subsumes [(1) user-generated explanation, as a list] One test subsumed another for the reason given by the argument, a programmer-generated explanation. See Section 9 for more details. There is another version of this inference (see immediately above).

tests-not-satisfied [(1) the tests not satisfied] The tests of the "subsumer" are not a subset of the tests of the "subsumee".

thing-doesnt-satisfy [(1) the **ALL** restriction not satisfied] `THING` cannot satisfy a non-vacuous **ALL** restriction.

transitive-closure-or-distribution [any number of (<**SAME-AS** restr> <role-path>) pairs, where the role-path is the role-path from the top-level object, along which the contributing **SAME-AS** restr was found] A concept or individual satisfies a **SAME-AS** restriction by a combination of transitive closure and distribution over **ALL** restrictions. For example, suppose concept `C1` is defined as:

```
(SAME-AS (lawyer) (best-friend)),
```

and concept `C2` is defined as:

```
(AND (SAME-AS (lawyer) (teacher))
     (SAME-AS (teacher) (best-friend))).
```

If we call (**cl-exp-subsumes-concept** `@c1 @c2`), we see that the **SAME-AS** restriction on `C1` is satisfied by **transitive-closure-or-distribution**, with the two **SAME-AS** restrictions: `((teacher) (best-friend))` and `((lawyer) (teacher))`.

8 Deciding What is Interesting About Objects

In practice, objects in **CLASSIC** often grow rather complex. Any accurate and useful domain model about real objects inevitably will be filled with many concepts, roles, rules, individuals and interconnections. A concept can have many instances, children, ancestors, rules and any of a number of restrictions. Similarly, an individual can have any number of restrictions, role fillers, primitive ancestors and so on. Printing an object or explaining some derived information about that object can take a lot of time and, when all is said and done, provide far more information that one can sift through in order to find whatever is of interest.

To address this, **CLASSIC** provides several powerful mechanisms for describing what aspects of an object are interesting and under what circumstances those aspects are interesting. When appropriate, this information is used to prune printed output so that only the information that is interesting is displayed.

These descriptions for what is interesting about a concept (and its instances) is stored on the concept's *meta individual* (see Section 2.4). These descriptions capture the meta information for what is interesting about that concept, and follow the normal rules for inheritance. If something is interesting about a concept (and its instances) under some circumstance, it is also interesting for all of its descendants (and their instances) under those circumstances.

8.1 Describing and Printing Interesting Aspects

The mechanisms for describing what is interesting about a class of objects are embedded in just a few functions. Those functions, as well as some functions that print or explain only interesting things, are described below. For a complete list of these functions, including some utility macros, see Appendix D.12.

8.1.1 Describing the Interesting Aspects of Objects

Facts about objects are either displayed or explained. The mechanisms that associate descriptions of what facts are interesting maintain this distinction. It is possible to have some fact be interesting to display but not interesting to explain or vice-versa.

cl-set-up-concept

One of the two main functions for describing interesting aspects is **cl-set-up-concept**. The syntax is

```
(cl-set-up-concept <symbol>
                   &key (group-names '(t)) (object-filter '(and))
                        (role-path NIL) (role-aspect NIL)
                        (toplevel-aspect NIL) (aspect-before-filter '(and))
                        (aspect-function '(identity))
                        (aspect-after-filter '(and)))
```

It takes the name of a **CLASSIC** concept and a set of values for describing what is interesting about that concept and the conditions under which those things are interesting. Individuals that are instances of this concept will only have the information that is described as interesting displayed when printed with **cl-print-ind-interesting**. Similarly, the concept itself will only have the information described as interesting displayed when printed with **cl-print-concept-interesting**. Subconcepts of this concept will also inherit these interesting properties.

There are two types of keyword arguments to **cl-set-up-concept**: those that describe *which aspects* of an object are interesting and those that describe *under which conditions* those aspects are interesting.

When describing what is interesting about a concept and its instances, one may specify either a toplevel-aspect or a role-aspect and a role-path. If toplevel-aspect is specified then it must be one of **ONE-OF**, **TEST**, **PRIM**, **MIN**, **MAX**, **SAME-AS**, **PARENTS**, **DESCENDANTS**, **INSTANCES**, **DESCENDANT-INSTANCES**, **CHILDREN**, **LOCAL-RULES**, **RULES**, or **COMMENT**.

Permissible role-aspects are **ONE-OF**, **TEST**, **PRIM**, **MIN**, **MAX**, **AT-LEAST**, **AT-MOST**, **ALL-NAMED-CONCEPT**, **FILLS**, **SAME-AS** and `T`. Each of these aspects describes some part of a role-restriction on an object. The role-aspect **ALL-NAMED-CONCEPT** can be used to assert that a role restriction is interesting when it is actually a named concept. The role-aspect value of `T` means simply "all role restrictions" and may be used as a shortcut to assert all of the other role-aspects, including **ALL-NAMED-CONCEPT**.

The keyword argument group-names allows one to "tag" an interesting description with arbitrary symbols or names. When printing out interesting things one can specify which descriptions to use by specifying these names.

Those keywords describing under which conditions an aspect is interesting are: object-filter, aspect-before-filter, aspect-after-filter and aspect-function. The object-filter is a **CLASSIC** expression. The value of an object's aspect will only be printed if the object can "pass" this filter (i.e. is subsumed by or is an instance of the **CLASSIC** expression). Once an aspect has been proven interesting, the values of that aspect (e.g. the list of parents or the fillers of a particular role) are considered. First, each value of the interesting aspect must pass the **CLASSIC** expression aspect-before-filter (much like an object must pass the object-filter). Those that do are then given to the aspect-function. An aspect-function is a lisp expression of the form (**function** $< arg_2 > < arg_3 > ... < arg_n >$)[18]. The aspect values are passed, as a list, as the first argument to the **function** with any additional arguments passed second, third and so on. **Function** will return the list or a subset of it. Those values must then, in turn, pass the aspect-after-filter.

Finally, those values that survive are printed out.

Note that the default values for each of the filters and the aspect-function are such that every aspect and aspect-value will pass.

Also note that repeated calls to **cl-set-up-concept** for a particular concept name and a particular aspect are combined: the object-filters are combined as if by an implicit **AND**. This is also true for the aspect-before-filters and the aspect-after-filters. Similarly, the aspect-functions are combined so as to produce their intersections. A concept inherits all the calls to **cl-set-up-concept** for all of its ancestors.

There are several ways that this function is useful. For example, it is reasonable to decide for a particular application that printing out all of the primitive ancestors of a concept or an individual is uninteresting and that, in fact, only the most specific primitive concepts are useful, in general. In this case, one could encode this at the top level:

```
(cl-set-up-concept 'classic-thing
                   :toplevel-aspect 'prim
                   :aspect-function 'cl-most-specific-concepts)
```

The aspect-function **cl-most-specific-concepts**, given a list of concepts, returns only those concepts from the list that have no children in the list. This description would be inherited by all the descendants of `CLASSIC-THING`. Thus, displaying `Mary` by invoking (**cl-print-ind** @i{`Mary`}) might yield, among other things:

```
Mary →
  ...
  Primitive Ancestors: (PERSON CLASSIC-THING)
  ...
```

On the other hand, displaying `Mary` by invoking (**cl-print-ind-interesting** @i{`Mary`}) would yield:

```
Mary →
  ...
```

[18]For a complete list of built-in **CLASSIC** functions as well as instructions for building special-purpose aspect functions, see Appendix D.12.4 and Appendix D.12.5, respectively

Primitive Ancestors: (PERSON)
...

This is because CLASSIC-THING subsumes PERSON. The PERSON concept *inherited* the call to **cl-set-up-concept** on CLASSIC-THING in the same way that PERSON inherits any restrictions placed on CLASSIC-THING. Any aspect, to be found interesting, must satisfy all of these restrictions. For example, imagine this series of calls to **cl-set-up-concept**:

```
(cl-set-up-concept 'classic-thing
                   :toplevel-aspect 'prim
                   :aspect-function 'cl-most-specific-concepts)
(cl-set-up-concept 'classic-thing
                   :toplevel-aspect 'comment)
(cl-set-up-concept 'person
                   :role-path @r{children}
                   :role-aspect 'fills)
(cl-set-up-concept 'student
                   :role-path @r{children}
                   :role-aspect 'fills
                   :aspect-function '(cl-member-of-concepts student))
(cl-set-up-concept 'chess-player
                   :role-path @r{children}
                   :role-aspect 'fills
                   :aspect-function '(cl-member-of-concepts chess-player))
```

First, we have said that we always care about comments and primitive ancestors. We have also indicated that the children of PERSONs are interesting in general; however, we have further indicated that for STUDENTs we are only interested in their children if they are also STUDENTs. Similarly, a CHESS-PLAYER's children are only interesting if they are CHESS-PLAYERs as well. If Mary is both a STUDENT and a CHESS-PLAYER then only those children of Mary's that are also *both* STUDENTs and CHESS-PLAYERs will be displayed (as opposed to all those children of Mary's that are *either* STUDENTs or CHESS-PLAYERs).

cl-set-up-concept-exp

The other main function for describing interesting aspects is **cl-set-up-concept-exp**. The syntax is

```
(cl-set-up-concept-exp <symbol>
                       &key (group-names '(t)) (object-filter '(and))
                            (role-path NIL) (role-aspect NIL)
                            (toplevel-aspect NIL) (aspect-before-filter '(and))
                            (aspect-function '(identity))
                            (aspect-after-filter '(and)))
```

This is the same as **cl-set-up-concept** except that it describes what facts are interesting to explain when **cl-print-exp-interesting** is called on an object.

cl-set-down-concept

The function **cl-set-down-concept** clears all calls to **cl-set-up-concept** made specifically to a concept. It is as if those calls were never made. The syntax is

(**cl-set-down-concept** <symbol>)

Note that this means that any descendants of this concept will "uninherit" these calls as well. On the other hand, this does *not* clear any calls to **cl-set-up-concept** inherited by the concept; those calls still exist.

cl-set-down-concept-exp

This clears calls to **cl-set-up-concept-exp** in exactly the same way that **cl-set-down-concept** clears calls to **cl-set-up-concept**. The syntax is

(**cl-set-down-concept-exp** <symbol>)

cl-copy-concept-set-up-to-exp

This procedure takes all the information describing what is interesting to print about a concept's children and instances *at the time of the function call* and also makes it interesting to explain. The syntax is

(**cl-copy-concept-set-up-to-exp** <symbol>)

8.1.2 Printing Interesting Aspects of Objects

Once the information that describes what things are interesting about certain objects have been set up, there are several ways to display or explain that information. Below we describe them.

cl-print-ind-interesting

This prints an individual as **cl-print-ind** except that it only prints out those facts about the individual deemed interesting as described by calls to **cl-set-up-concept**. The keyword interesting-groups allows further control over which of those descriptions are important. The syntax is

cl-print-ind-interesting <individual>
&key (interesting-groups `'(t)`) (told `NIL`))

cl-print-concept-interesting

This prints a concept as **cl-print-concept** except that it only prints out those facts about the concept that are determined to be interesting as described by calls to **cl-set-up-concept**. The syntax is

(**cl-print-concept-interesting** <individual>
&key (interesting-groups `'(t)`) (told `NIL`)
(all `'unspecified`))

The keyword interesting-groups allows additional control over which descriptions are important. As noted in the description for **cl-set-up-concept**, descriptions can be tagged with arbitrary names. The interesting-groups keyword allows you to specify which descriptions to use for determining interestingness by listing their names. For example, one might label some descriptions with the symbol 'debugging and ask for these descriptions when debugging but use any descriptions labelled with the symbol 'demo for talks and presentations.

cl-print-exp-interesting

This prints the explanation of an object as **cl-print-exp** except that it only explains those facts deemed interesting as described by calls to **cl-set-up-concept-exp**. This function performs an implicit call to **cl-clear-exp**. The keyword interesting-groups allows further control over which of those descriptions are important. The syntax is

(**cl-print-exp-interesting** <individual> &key (interesting-groups `'(t)`))

8.1.3 Interactions with Pruning Variables

The traditional method in **CLASSIC** for pruning printed information has been to use the pruning variables described in Section 7.2. It is still permissible to use these pruning variables while describing what is interesting about objects using meta information as outlined in this section; however, it is possible for pruning variables to interact with meta information in undesirable ways. For example, imagine that you have decided that **AT-LEAST** information is interesting for the role `important-role`:

(**cl-set-up-concept-exp** 'classic-thing
:role-aspect 'atleast
:role-path @r{important-role})

but have set the pruning variable ***cl-exp-ignore-at-least*** to `T`. Setting the pruning variable in this way overrides the description above. This is because it is equivalent to declaring for every role in the knowledge base something like:

(**cl-set-up-concept-exp** 'classic-thing
:object-filter 'always-false
:role-aspect 'atleast
:role-path *role*)

where the function **always-false** always returns false. Recall that meta information about interesting aspects are always combined: repeated calls to **cl-set-up-concept-exp** for a particular concept name and a particular aspect cause the object-filters to be combined as if by an implicit **AND**. This means that for every role in the database, no object will ever pass the object filter for the role-aspect `at-least`. Thus, the **AT-LEAST** restriction on important-role will never be explained.

On the other hand, pruning variables can provide control over procedural pruning such as where to stop on complete explanation (e.g. by using the pruning variable ***cl-complete-exp-ignore-rules***). In short, pruning variables and meta information can complement one another. A reasonable and safe approach to avoiding problems is to turn off all pruning due to pruning variables:

```
(*cl-exp-set-pruning-variables* NIL)
```

and then selectively switch on those pruning variables that seem to be needed in general for the application in question and cannot be easily duplicated using meta information.

8.2 Printing and Explaining Interesting Descriptions

With a facility for describing things that are interesting comes the need to examine those descriptions. For example, it becomes important to understand why something you thought was interesting turned out not to be (or why something you thought was uninteresting turned out to be interesting). This is useful for debugging the descriptions themselves, of course, but it can also be a useful tool for further understanding the domain hierarchy. In this section, we describe functions used for printing and explaining interesting descriptions.

cl-print-all-interesting-descriptions

This function prints the interesting descriptions that apply to an object. Basically, this is a way to recover the calls made to **cl-set-up-concept** and to determine how they have combined to examine the interesting aspects of a particular object. The syntax is

```
(cl-print-all-interesting-descriptions <object>
                                       &key (skip-filtering NIL) (annotate NIL)
                                            (interesting-groups '(t)))
```

Let us return to our earlier example:

```
(cl-set-up-concept 'classic-thing
                   :toplevel-aspect 'prim
                   :aspect-function 'cl-most-specific-concepts)
(cl-set-up-concept 'classic-thing
                   :toplevel-aspect 'comment)
(cl-set-up-concept 'person
                   :role-path @r{children}
                   :role-aspect 'fills)
(cl-set-up-concept 'student
                   :role-path @r{children}
                   :role-aspect 'fills
                   :aspect-function '(cl-member-of-concepts student))
(cl-set-up-concept 'chess-player
```

```
            :role-path @r{children}
            :role-aspect 'fills
            :aspect-function '(cl-member-of-concepts chess-player))
```

So, if `Mary` is just a `PERSON`, (**cl-print-all-interesting-descriptions** @i{`Mary`}) might print:

```
    Mary →
...
      description:
        groups:          (T)
        role-path:       @r{CHILD}
        role-aspect:     FILLS
...
```

On the other hand, if `Mary` is just a `STUDENT`, (**cl-print-all-interesting-descriptions** @i{`Mary`}) might print:

```
    Mary →
...
      description:
        groups:          (T)
        role-path:       @r{CHILD}
        role-aspect:     FILLS
        aspect-function: (CL-MEMBER-OF-CONCEPTS STUDENT)
...
```

Later, if `Mary` joins the chess club, what is interesting about her changes as well:

```
    Mary →
...
      description:
        groups:          (T)
        role-path:       @r{CHILD}
        role-aspect:     FILLS
        aspect-function: (CL-INTERSECTION
                             (CL-MEMBER-OF-CONCEPTS STUDENT)
                             (CL-MEMBER-OF-CONCEPTS CHESS-PLAYER))
...
```

The aspect-function **cl-intersection** serves the same purpose as **AND** does in **CLASSIC** expressions.

These examples demonstrate **cl-print-all-interesting-descriptions**' default behaviour. It is possible to display different information by providing values to the keyword arguments. For example, if the keyword annotate is non-`NIL` then a description of how a particular description was derived for an aspect is also printed:

```
Mary →
...
    description:
              groups:         (T)
              role-path:      @r{CHILD}
              role-aspect:    FILLS
              aspect-function: (CL-INTERSECTION
                                (CL-MEMBER-OF-CONCEPTS STUDENT)
                                (CL-MEMBER-OF-CONCEPTS CHESS-PLAYER))
    annotations:
      from: (TOLD-INTERESTING)
              aspect-function: (CL-MEMBER-OF-CONCEPTS STUDENT)
      from: (TOLD-INTERESTING)
              aspect-function: (CL-MEMBER-OF-CONCEPTS CHESS-PLAYER)
      from: (INHERITANCE @c{CLASSIC-THING})
...
```

In this case, **CLASSIC** was *told* that `STUDENT`s are interesting `children` fillers for `STUDENT`s and that `CHESS-PLAYER`s are interesting `children` fillers for `CHESS-PLAYER`s. Finally, `Mary` inherited from `CLASSIC-THING` the description that fillers for children is interesting.

It is also possible to display more information by setting the skip-filtering keyword to a non-`NIL` value. Usually, any interesting descriptions that fail because of the object-filter are not displayed. Providing a non-`NIL` value for skip-filtering displays these descriptions as well (*Note:* If annotate is `t` then skip-filtering is ignored; all descriptions will be printed regardless of its value).

The last keyword is interesting-groups. As noted above, descriptions can be tagged with arbitrary names. Using the interesting-groups keyword allows you to specify that you are interested only in descriptions with particular names.

cl-print-all-interesting-descriptions-exp

This function is like **cl-print-all-interesting-descriptions** except that it recovers calls to **cl-set-up-concept-exp**. The syntax is

```
(cl-print-all-interesting-descriptions-exp <object>
                                           &key (skip-filtering NIL)
                                                (annotate NIL)
                                                (interesting-groups '(t)))
```

cl-print-interesting-description

This function is like **cl-print-all-interesting-descriptions** except that it only prints the interesting description for a particular toplevel-aspect or for a particular role-aspect along a given role-path. The syntax is

```
(cl-print-interesting-description <object>
```

```
&key (annotate NIL) (interesting-groups '(t))
     (role-path NIL) (role-aspect NIL)
     (toplevel-aspect NIL))
```

cl-print-exp-interesting-description-not-printed

This function explains why a particular object (or list of objects) that acts as a filler of either a toplevel-aspect or as a filler of a role-aspect along a given role-path is not considered interesting enough to print. If aspect-fillers is `NIL` then all possible fillers are explained. The syntax is

```
(cl-print-exp-interesting-description-not-printed
   <object>
   &key (interesting-groups '(t)) (stream t)
        (aspect-fillers NIL) (role-path NIL)
        (role-aspect NIL) (toplevel-aspect NIL))
```

Let us return to our example above with our student-chess player `Mary`. Assume that she has a child `Bob` who is student, `Fred` who is a chess-player and another `Sue` who is both. Invoking:

```
(cl-print-exp-interesting-description-not-printed @i{Mary}
                                                   :role-path @r{children}
                                                   :role-aspect 'fills)
```

yields an annotation explaining why `Bob` and `Fred` are not printed as fillers for `children` after a call to (**cl-print-ind-interesting** @i{`Mary`}):

```
Mary →
  description:
    groups:          (T)
    role-path:       @r{CHILD}
    role-aspect:     FILLS
    aspect-function: (CL-INTERSECTION
                       (CL-MEMBER-OF-CONCEPTS STUDENT)
                       (CL-MEMBER-OF-CONCEPTS CHESS-PLAYER))
  annotations:
    from: (TOLD-INTERESTING)
                     failure-point: FAILED-ASPECT-FUNCTION
                     failed-on-aspect-fillers: (@i{Bob} @i{Fred})
```

As one would expect, both individuals failed to pass the aspect-function. They could have also failed the aspect-before-filter or the aspect-after-filter. It is also possible that `Mary` might not have passed the object-filter (in which case, no information would have been printed with **cl-print-ind-interesting**) or that the role-aspect FILLS for the role `children` wasn't described as interesting at all.

Finally, note what would happen if we asked why `Sue` was not printed:

```
(cl-print-exp-interesting-description-not-printed @i{Mary}
                                                  :role-path @r{children}
                                                  :role-aspect 'fills
                                                  :aspect-fillers @i{Sue})
```

In this case, one would see:

```
Mary →
  description:
    groups:          (T)
    role-path:       @r{CHILD}
    role-aspect:     FILLS
    aspect-function: (CL-INTERSECTION
                          (CL-MEMBER-OF-CONCEPTS STUDENT)
                          (CL-MEMBER-OF-CONCEPTS CHESS-PLAYER))
  annotations:
    from: (TOLD-INTERESTING)
                     failure-point: NO-FAILURE-POINT
                     failed-on-aspect-fillers: @i{Sue}.
```

Since `Sue` would asctually be displayed there is no failure point.

8.3 Inferences for Interesting Descriptions

As with classification in general, **CLASSIC** performs different kinds of inferences in order to decide what is interesting and what is not. It is possible to get access to the inferences used to decide that a particular aspect will be displayed (or explained) as well as to the inferences that were used to decide that a particular aspect will not be displayed (or explained). These inferences are usually accessed through **cl-print-all-interesting-descriptions**, **cl-print-all-interesting-descriptions-exp**, **cl-print-interesting-description**, and **cl-print-exp-interesting-descriptions-not-printed**. Below is a list of those inferences. In these descriptions of the inferences, the arguments to each inference are listed after the inference type, enclosed within [ ]'s. [ ] means no arguments.

8.3.1 Inferences for Displaying An Apsect

These inferences are used to determine how a particular description is derived for a given object.

told-interesting [ ] If part of a description is set up on a concept then it is simply thought of as told information.

inheritance [(1) a told concept from which the description is inherited] This is the same as the regular inference `inheritance`. For object `A`, if `A` is a subclass or instance of `B`, and `B` has the interesting description C, then `A` inherits C from `B`.

8.3.2 Inferences for Not Displaying An Apsect

An aspect filler can fail to be interesting at any of several points.

failed-object-filter [ ] If an object fails the object-filter, none of its aspect fillers will be displayed or explained.

failed-aspect-before-filter [ ] If an aspect filler of an object fails the aspect-before-filter for that aspect, it will not be displayed or explained.

failed-aspect-function [ ] If the aspect-function for an aspect does not return a particular aspect filler, it will not be displayed or explained.

failed-aspect-after-filter [ ] If an aspect filler of an object fails the aspect-after-filter for that aspect, it will not be displayed or explained.

not-an-aspect-filler [ ] If a given value never filled an aspect for another, it cannot be displayed or explained as an aspect filler for that object.

not-told-interesting [ ] If an aspect was never described as interesting, none of the fillers of that aspect will be displayed or explained.

no-failure-point [ ] If an aspect filler is indeed interesting then there was no failure point and asking for the point of failure returns this inference.

9 Extending CLASSIC Using Tests

NOTE: this section is for advanced users. The features described in this section are semi-experimental. Their functionality will continue to exist in future versions of CLASSIC *but it is possible that the interface will change slightly.*

Test restrictions provide a way to use procedures to specify concepts (for a complete description of test restrictions, see Section 2.1.4.1). This is useful for a number of reasons. First, it allows CLASSIC to express concepts that it cannot with its other operators. Second, it allows CLASSIC to manipulate structures that it could not easily do otherwise.

For example, CLASSIC is often used by other systems as an embedded database and classifier. Although CLASSIC individuals and concepts are the expected method of data exchange between CLASSIC and the larger system, there may be data structures that are inconvenient to represent and reason with in this way.

For example, imagine a system that deals with some kind of structured data, such as text messages. This data has text and date information associated with it. The larger system wants to use CLASSIC to reason about incoming data in ways that depend upon the text and date information. In particular, the system needs some way to classify data based on whether particular strings appear as substrings in the text part of the data and whether dates in the date part of the data fall within certain ranges.

It is not straightforward to do this directly in the CLASSIC system. For example, while there is a CLASSIC operator that answers the question of whether a particular string appears as a filler of a role (i.e. `(FILLS` *role* `"data")`), there is no operator that answers the question of whether a particular string appears as a substring of one of the fillers of a role. Similarly, CLASSIC provides useful operators for testing whether integers fall within particular ranges; however, there is no similar operator for dealing with a structured set of integers, such as dates.

It is, of course, impossible for CLASSIC or any system to anticipate the needs of each and every user and provide built-in operators for dealing with every esoteric data structure one could imagine. Using test restrictions, it is easy to provide operators that answer the questions above. For example, here is the definition for a new CLASSIC operator **fills-substring**:

```
(defun fills-substring (ind role-name string)
   (let* ((r (cl-named-role role-name))
          (fillers (cl-fillers ind r))
          (ret (find-if
                  #'(lambda (filler)
                       (AND (stringp filler)
                            (case-insensitive-search
                                 string filler))
                  fillers))))
      (cond (ret t)
            ((cl-ind-closed-role? ind r) nil)
            (t '?))))
```

This function returns `T` if and only if there exists some filler of `role-name` that contains `string` as a substring. One might use it in a CLASSIC expression like this:

```
(AND (TEST-C fills-substring text-sender "accounting")
      ...)
```

We are now able to specify concepts involving substrings. In fact, CLASSIC can even do some further reasoning about concepts using **fills-substring**. For example, imagine two concepts:

```
CONCEPT1:  (AND (AT-LEAST 1 text-sender)
                (TEST-C fills-substring text-sender "accounting"))

CONCEPT2:  (AND (AT-LEAST 1 text-sender)
                (TEST-C fills-substring text-sender "accounting")
                (TEST-C fills-substring text-receiver "research"))
```

The first concept subsumes the second. CLASSIC can deduce this easily because the expressions that make up `CONCEPT1` are a subset of those that make up `CONCEPT2`.

Still, **fills-substring** is a black box. CLASSIC has no understanding of the semantics (intended or actual) of such test functions. As a result, it isn't possible for CLASSIC to do any but the kind of trivial textual reasoning described above. For example, imagine these two concepts:

```
CONCEPT3:  (AND (AT-LEAST 1 text-sender)
                (TEST-C fills-substring text-sender "account"))

CONCEPT4:  (AND (AT-LEAST 1 text-sender)
                (TEST-C fills-substring text-sender "accounting"))
```

Again, the first concept subsumes the second. Certainly, if the role `text-sender` has a filler that includes "accounting" as a substring then the same filler also includes "account" as a substring. Unfortunately, CLASSIC cannot perform this kind of reasoning; it doesn't understand the meaning of **fills-substring**.

In this section, we describe several mechanisms for allowing CLASSIC to reason with test restrictions. When completely described with these mechanisms, tests can act much more like full-fledged CLASSIC operators, participating in complex reasoning about subsumption, conjunction, normalization, consistency and implication.

Throughout the next subsections, we will illustrate how to use these mechanisms. For this discussion, we will assume the existence of four test functions:

1. **fills-substring**. This is the function defined above. It takes two arguments, a role and a string. It returns true if and only if there exists some filler of the role that contains the string as a substring.

2. **date-before**. This function takes four arguments, an attribute and three integers representing a year, a month and a day, respectively. This function returns true if and only if the attribute contains a date object (some user-created structure in LISP) that is on or before `year`, `month` and `day`.

   One would use **date-before** in a CLASSIC expression like this:

   ```
   CONCEPT5:  (AND (AT-LEAST 1 arrival)
                   (AT-MOST 1 arrival)
                   (ALL arrival DATE-OBJECT)
                   (TEST-C date-before arrival 1995 9 2)
                   ...)
   ```

   Assuming that `arrival` is a valid role and `DATE-OBJECT` is a valid concept, the class `CONCEPT5` has one filler for its `arrival` role, the filler is a `DATE-OBJECT` and instances of `CONCEPT5` must satisfy the test that any date structure that fills `arrival` must represent a day that is on or before September 2, 1995.

3. **date-after**. This function takes four arguments, an attribute and three integers representing a year, a month and a day, respectively. This function returns true if and only if the attribute contains a date object that happens on or after `year`, `month` and `day`.

   One would use **date-after** in a CLASSIC expression like this:

   ```
   CONCEPT6:  (AND (AT-LEAST 1 arrival)
                   (AT-MOST 1 arrival)
   ```

```
                (ALL arrival DATE-OBJECT)
                (TEST-C date-after arrival 1995 9 2)
                ...)
```

The class CONCEPT6 has one filler for its arrival role, the filler is a DATE-OBJECT and instances of CONCEPT6 must satisfy the test that any date structure that fills arrival must represent a day that is on or after September 2, 1995.

4. **date-between**. This function takes seven arguments, an attribute, three integers representing a year, a month and a day, and another three integers representing a year, a month and a day. This function returns true if and only if the attribute contains a date object that happens between the two dates, inclusive.

   One would use **date-between** in a CLASSIC expression like this:

```
CONCEPT7:  (AND (AT-LEAST 1 arrival)
                (AT-MOST 1 arrival)
                (ALL arrival DATE-OBJECT)
                (TEST-C date-between arrival 1995 9 1 1995 9 3)
                ...)
```

   The class CONCEPT7 has one filler for its arrival role, the filler is a DATE-OBJECT and instances of CONCEPT7 must satisfy the test that any date structure that fills arrival must represent a day that is between September 1 and September 3, 1995.

It is worth noting that whenever test functions are used in normal concept expressions, the **TEST-C** or **TEST-H** keyword must be included; however, for internal processing, CLASSIC removes the **TEST-C** and **TEST-H** keywords. Because they mimic internal CLASSIC processing, the mechanisms described in the sections below also follow this convention. Therefore *test expressions* have no **TEST-C** or **TEST-H** keyword. For example, the expression (**fills-substring** r "abc") is a valid test expression while (**TEST-C fills-substring** r "abc") is not. By contrast, when entering a concept expression, (**fills-substring** r "abc") would be an invalid concept expression while (**TEST-C fills-substring** r "abc") would be perfectly valid.

9.1 Test Normalization

Normalization for tests is the process of taking an expression in one form and turning it into another form that is easier to represent and work with. For example, imagine that our three date constructors are used within a larger system that can most easily provide date information in the form of strings, such as "January 12, 1992" and "8/6/91". We might want to allow this system to assert (**date-after** arrival "January 12, 1992") and then normalize the expression into (**date-after** arrival 1992 1 2). We might also want to take the opportunity to note that (**date-after** arrival "January 42 1991") is not a well-formed expression because there is no forty-second day of January.

You are able to specify this kind of reasoning with a *test normalization function*. It takes one argument: a (possibly unnormalized) test expression. The normalization function should return two arguments (using LISP's (**values**) form). The first is the normalized form of the test expression or `NIL` if the expression causes an error (e.g. is ill-formed). The second is an explanation of what happened. The explanation should be a list: an inference rule followed by any arguments.

If the first return value is non-`NIL`, the second value will be ignored during normal CLASSIC processing. If the first return value is `NIL` and no second value is returned, CLASSIC will generate an explanation for you; however, it may not be as instructive as you would like.

For example, **fills-substring** takes a role name and a string. This function verifies that it is being used correctly:

```
(defun fills-substring-normalize (expr)
    (cond ((not (= (length expr) 3))
           (values nil '(incorrect-number-of-args-error ,@expr)))
          ((not (cl-named-role (nth 1 expr)))
           (values nil (list 'not-a-role-error (nth 1 expr))))
          ((not (stringp (nth 2 expr)))
           (values nil (list 'not-a-string-error (nth 2 expr))))
          (t expr)))
```

This function will signal an error if **fills-substring** is not passed exactly two arguments or if those arguments are of the wrong types. In this function, an error will include a useful explanation with an inference rule and arguments that explain the source of the error. Note that in the above example, we have used actual CLASSIC error codes. See Appendix E for a description of all CLASSIC error codes. If one of these error codes is appropriate for your test normalization function, and takes the right arguments, you should use it; otherwise, you should create your own error codes.

The function **cl-define-normalize-rule-by-function** registers normalization functions. The syntax is

(**cl-define-normalize-rule-by-function** <symbol> <function>)

For example, we might register our normalization function above like this:

```
(cl-define-normalize-rule-by-function
    fills-substring #'fills-substring-normalize)
```

This expression registers this normalization function as one that normalizes test expressions using **fills-substring**. By convention, the normalization function for a test function named *test* should be named *test*`-normalize`.

CLASSIC will use the normalization functions whenever it is trying to normalize a test expression. Without a specific normalization function, CLASSIC can only check to see that the symbol used in a test expression as a test-function is actually bound to a function.

When testing a normalization function, it may be useful to use the function **cl-test-normalize**. The syntax is

(**cl-test-normalize** <test expression>)

The function takes a test expression, finds the registered normalization function, calls it on the test expression, and returns the normalized test expression. If no normalization function exists, the original test expression is returned.

It is worth mentioning that CLASSIC only keeps track of the normalized test expression resulting from this step. This has two major implications. First, all of the mechanisms described below will only see normalized test expressions. Therefore, it is not necessary for them to perform most basic syntax error checking if this is done in the normalization function.

Second, the original CLASSIC expression typed in by the user may be lost. For example, if both (**date-after** arrival "January 1, 1992") and (**date-after** arrival "New Year's Day, 1992") become (**date-after** arrival 1992 1 1) during the normalization process, it will not be possible to determine which expression was originally provided. If the programmer wishes to keep this information available (e.g. in order to print test expressions in their original forms), it must be done manually.

One common way to keep the original "told" form around is to add an element to the test expression. For example, normalization might turn (**date-after** arrival "January 1, 1992") into (**date-after** arrival 1992 1 1 "January 1, 1992") while turning (**date-after** arrival "New Year's Day, 1992") into (**date-after** arrival 1992 1 1 "New Year's Day, 1992"). Generally, operations such as subsumption would ignore this extra argument, but it would always be there when information is to be presented to the user.

It is also possible to simply ignore the original unnormalized expression and treat the output of the normalization function as the told form. In any case, the programmer has full freedom to decide how to track the told form.

9.2 Test Subsumption

In our example above:

```
CONCEPT3:  (AND (AT-LEAST 1 text-sender)
                (TEST-C fills-substring text-sender "account"))

CONCEPT4:  (AND (AT-LEAST 1 text-sender)
                (TEST-C fills-substring text-sender "accounting"))
```

we noted that CONCEPT3 subsumes CONCEPT4. In order to allow CLASSIC to perform this kind of reasoning, one specifies the subsumption relationship with a *test subsumption function*. The function takes two test expressions as arguments and returns two arguments. The first return value is non-NIL if and only if the first expression subsumes the second. The second is an explanation of what happened. The explanation should be a list: an inference rule followed by any arguments. For example, this function determines subsumption between two **fills-substring** test expressions.

```
(defun fills-substring-subsumes (expr1 expr2)
    (let ((r1 (nth 1 expr1))
          (r2 (nth 1 expr2))
          (str1 (cddr expr1))
          (str2 (cddr expr2)))
      (if (cl-named-role-parent-of?  r1 r2)
          (if (case-insensitive-search str1 str2)
```

```
                t
                (values nil (list 'not-a-substring str1 str2)))
            (values nil (list 'not-an-ancestor-role r1 r2)))))
```

where `cl-named-role-parent-of?` takes two role names and returns `T` if the first role name is the same as the second or an ancestor role and `case-insensitive-search` returns `T` when its first argument is a substring of its second argument.

A **fills-substring** expression subsumes another when two conditions are met: 1) the first's role is at least as general as the second's and 2) the string in the first expression is a substring of the string in the second. For example:

```
(fills-substring-subsumes
      '(fills-substring name "Tom")
      '(fills-substring name "Kirk"))
   → NIL
     (NOT-A-SUBSTRING "Tom" "Kirk")

(fills-substring-subsumes
      '(fills-substring name "Tom")
      '(fills-substring name "T"))
   → NIL
     (NOT-A-SUBSTRING "Tom" "T")

(fills-substring-subsumes
      '(fills-substring name "T")
      '(fills-substring name "Tom"))
   → T

(fills-substring-subsumes
      '(fills-substring name "T")
      '(fills-substring friend "Tom"))
   → NIL
     (NOT-AN-ANCESTOR-ROLE NAME FRIEND)
```

The last example assumes that `name` is not an ancestor role of the role `friend`.

The test expressions passed to a subsumption function do not have to use the same test function. We might set up a subsumption relationship between **date-before** and **date-between**. We would simply define a function that expects a **date-before** expression and a **date-between** expression and checks to make certain that the roles are compatible and that the date of the **date-before** expression is on or after the second date for **date-between**.

Usually, there is no need for any error checking on the expressions. CLASSIC will have already used the normalization functions that have been provided to make certain that both expressions are syntactically correct.

The function **cl-define-subsumption-rule-by-function** registers subsumption functions. The syntax is

(**cl-define-subsumption-rule-by-function** <symbol> <symbol> <function>)

For example, we might register our function that determines subsumption for **date-before** and **date-between** like this:

```
(cl-define-subsumption-rule-by-function
    'date-before 'date-between
    #'date-before-subsumes-date-between)
```

The order of the symbols is important. This expression registers this subsumption function as one that checks whether **date-before** expressions subsume **date-between** expressions, not the other way around. Because of this, the subsumption function can always expect that the first expression will be a **date-before** expression and the second a **date-between** expression. By convention, a subsumption function for $testname_1$ and $testname_2$ should be named $testname_1$`-subsumes-`$testname_2$. If the two tests are the same, the convention allows the function to be named *testname*`-subsumes`.

CLASSIC will use the subsumption functions whenever it is trying to determine whether one test subsumes another. Without such functions, CLASSIC can only deduce that one test expression subsumes another when both expressions are textually identical.

When testing a subsumption function, it may be useful to use the function **cl-test-subsume?**. The syntax is

(**cl-test-subsume?** <test expression> <test expression>)

The function takes two test expressions, finds their registered subsumption function, if one exists, and calls it on the test expressions. If no subsumption function exists, then **cl-test-subsume?** checks for textual equality. The function **cl-test-lists-subsume?** may also be useful. Please see Appendix D.13.2 for a complete description.

9.3 Test Conjunction

Often, it is the case that tests interact when they are *conjoined*. There are several kinds of interactions that two tests may have. In this section we will describe mechanisms for telling CLASSIC about two of them; namely, incompatibility and simplification.

Test expressions assert constraints about instances of a concept. Test incompatibility arises when different test expressions assert disjoint constraints. For example, consider three concepts:

```
CONCEPT8:  (date-between arrival 1995 1 1 1995 1 31)

CONCEPT9:  (date-between arrival 1995 12 1 1995 12 31)

CONCEPT10: (date-between arrival 1995 11 15 1995 12 15)
```

The first concept asserts that any date filling the `arrival` role must fall within January of 1995, the second asserts that any date filling the `arrival` role must fall within December of 1995 and the third asserts that any date filling the `arrival` role must fall between November 15^{th} and December 15^{th} of 1995. All three concepts are perfectly valid; however, this concept:

```
CONCEPT11:  (AND CONCEPT8 CONCEPT9)
```

cannot possibly have any instances. It isn't possible to arrive in both January and December. Therefore, CONCEPT8 and CONCEPT9 are incompatible and CONCEPT11 is incoherent.

On the other hand this concept:

```
CONCEPT12:  (AND CONCEPT9 CONCEPT10)
```

is valid because it is possible to arrive between November 15^{th} and December 15^{th} *and* during December, so long as you arrive between December 1^{st} and December 15^{th} 1995. In fact, these statements are equivalent:

```
(AND (date-between arrival 1995 12 1 1995 12 31)
     (date-between arrival 1995 11 15 1995 12 15))

(date-between arrival 1995 12 1 1995 12 15)
```

We might prefer the second, however. It is more compact and less expensive to reason with (for both CLASSIC and humans). This is simplification.

Incompatibility is really a special case of simplification (two tests are simplifying into the incoherent concept). Therefore, one specifies both kinds of conjunction relationships with a *test conjunction function*. Like subsumption functions, conjunction functions take two test expressions as arguments.

Conjunction functions should return two values. The first value is either: a new test expression if the two input test expressions conjoin and simplify, T if the two expressions are consistent but do not simplify, or NIL if the two expressions are incompatible together. The second value is an explanation of what happened, as a list.

If the first return value is T, the second value will be ignored. If the first value is not T then *something* happened (either a simplification or an inconsistency). Therefore, CLASSIC will use the second value as an explanation. If the first return value is not T and no second value is returned, CLASSIC will generate an explanation for you; however, it may not be as instructive as you would like.

Let us think about a conjunction function for conjoining **date-between** test expressions:

```
(defun conjoin-date-betweens (nlist1 nlist2)
  (let* ((d11 (apply #'convert-to-date (subseq nlist1 2 5)))
         (d12 (apply #'convert-to-date (subseq nlist1 5 8)))
         (d21 (apply #'convert-to-date (subseq nlist2 2 5)))
         (d22 (apply #'convert-to-date (subseq nlist2 5 8)))
         (r1 (cl-named-role (cadr nlist1)))
         (r2 (cl-named-role (cadr nlist2)))
         (pr (pick-descendant r1 r2))
         (dt-overlap (get-date-overlap d11 d12 d21 d22)))
    (cond ((not pr) t)
          ((not dt-overlap)
           (values nil '(date-ranges-do-not-overlap)))
          (t (normalize-date-between
                (append (list 'date-between (cl-role-name pr))
                        (date-to-numbers (car dt-overlap))
                        (date-to-numbers (cdr dt-overlap))))))))
```

The functions **convert-to-date** and **date-to-numbers** convert between the representation of dates used in the test expressions and the representation actually stored in roles. The function **pick-descendant** takes two roles and returns the one that is most specific in the role-hierarchy or NIL if they are unrelated. The function **get-date-overlap** returns the intersection of two date ranges.

If the roles used by the two test-expressions are unrelated then the two expressions can co-exist but do not interact. On the other hand, if they refer to related roles, they must interact. If the intersection of their date ranges is empty, then an inconsistency is signaled. In addition, an explanation is generated indicating that this is the case. For this example, the default CLASSIC explanation (that the two given test expressions were inconsistent) might be sufficient because there can only be one reason two **date-between** expressions are inconsistent; however, we have chosen to provide the explanation explicitly. By contrast, if two **date-between** expressions simplify, it is sufficient to simply accept the default explanation that will be generated; namely that the two test expressions resulted in a simpler expression (actually, the default explanation is a little smarter than this, see Section 9.6.1).

```
(conjoin-date-betweens
    '(date-between arrival 1995 1 1 1995 12 31)
    '(date-between arrival 1995 6 1 1996 6 1))
    → '(date-between arrival 1995 6 1 1995 12 31)

(conjoin-date-betweens
    '(date-between arrival 1995 1 1 1995 12 31)
    '(date-between arrival 1996 1 1 1996 12 31))
    → NIL
      (DATE-RANGES-DO-NOT-OVERLAP)

(conjoin-date-betweens
    '(date-between arrival 1995 1 1 1995 12 31)
    '(date-between departure 1996 1 1 1996 12 31))
    → T
```

The function **cl-define-conjoin-rule-by-function** registers conjunction functions. The syntax is

(**cl-define-conjoin-rule-by-function** <symbol> <symbol> <function>)

For example, we might register a function that determines conjunction between **date-before** and **date-between** like this:

```
(cl-define-conjoin-rule-by-function
    'date-before 'date-between
    #'conjoin-date-before-date-between)
```

As with subsumption, the order of the symbols is important. The expression registers **conjoin-date-before-date-between** as a function that conjoins **date-before** expressions with **date-between** expressions, not the other way around. Because of this, the conjunction function can always expect that the first expression will be a **date-before** expression and the second a **date-between** expression. By convention, such conjunction functions should be named `conjoin-`*testname*$_1$`-`*testname*$_2$. If the two tests are the same, the convention allows the function to be named `conjoin-`*testname*`s`.

CLASSIC will use conjoin functions whenever it is trying to merge new test expressions into a description. Without such functions, CLASSIC can only deduce that there is no need to keep two textually identical test expressions.

When testing a conjunction function, it may be useful to use the function **cl-test-conjoin**. The syntax is

(**cl-test-conjoin** <test expression> <test expression>)

The function takes two test expressions, finds their registered conjunction function, if one exists, and calls it on the test expressions. If no conjunction function exists, then `T` is returned.

9.4 Test Consistency

One of the things that CLASSIC does in its processing is determine whether a given object has become incoherent. Sometimes a given test is not consistent with the rest of an expression. For example, this expression is incoherent:

```
CONCEPT14:  (AND (ALL arrival DATE-OBJECT)
                 (AT-LEAST 1 arrival)
                 (AT-MOST 1 arrival)
                 (TEST-C date-between arrival 1995 9 1 1995 9 1)
                 (FILLS arrival September 12 1995)
                 ...)
```

The **date-between** implies that `arrival` contains the date September 1, 1995 while the **FILLS** asserts that `arrival` contains the date September 12, 1995. Because `arrival` can only have one filler, no object can satisfy this description.

There are two ways that CLASSIC might notice this inconsistency. In the case that an implication function has been provided[19], the **date-between** implication function could assert that the filler of `arrival` must be September 1. When CLASSIC adds the new information, it will notice the inconsistency as a part of its normal processing.

On the other hand, recall that test functions are expected to return `T`, `'?` or `NIL`. `T` means that the given individual satisfies and will always satisfy this test. `'?` means that the individual does not currently satisfy the test but may later. `NIL` means that the individual will never satisfy this test.

For example, the test function **date-between** will return `NIL` on an individual only in the case that the role it is given as an argument is closed and has no filler that satisfies the date represented by its other arguments.

As a part of its inconsistency checking on a description, CLASSIC can run some of the description's tests. Because tests are run on individuals, CLASSIC constructs a *temporary individual*. A temporary individual is an individual that is not a part of the knowledge

[19]Implication functions are not currently provided, but will be soon (see Section 9.7.4).

base. In this case, it is constructed to represent an individual that meets the minimal requirements for the description being tested for inconsistency. In general, the instances of a concept will contain far more information than is required to be an instance of the concept. For example, consider this concept:

```
CONCEPT15:  (date-between arrival 1995 1 1 1995 1 31)
```

The concept `CONCEPT15` may have many instances. Some of these instances may have other restrictions on the `arrival` role or may have restrictions on other roles as well. On the other hand, an individual that meets the minimal requirements for `CONCEPT15` would have only the fact `(`**date-between** `arrival 1995 1 1 1995 1 31)` asserted about it. This is the kind of individual CLASSIC constructs.

If one of a concept's tests is run on a minimal individual for that concept and it returns `NIL`, it must be the case that *no* individual can possibly satisfy the concept. The concept is, therefore, inconsistent.

Let us return to `CONCEPT14`. If the test `(`**TEST-C date-between** `arrival 1995 9 1 1995 9 1)` is executed on the minimal individual for this description, it should return `NIL`, for the reasons mentioned at the beginning of this section.

Because test expressions themselves are used for consistency checking, there is no need to write and register a consistency testing function. On the other hand, because some test functions are ill-behaved or can affect the knowledge base, CLASSIC has to be told to perform this check. This is done with the function **cl-set-test-for-incompatible-test**. The syntax is

(**cl-set-test-for-incompatible-test** <symbol>)

We would tell CLASSIC to use **date-between** for incompatibility testing like this:

(**cl-set-test-for-incompatible-test 'date-between**)

CLASSIC tests can normally return three values (see Section 2.1.4.1). If a test will be used for incompatibility testing, it can return a fourth. The second and third are ignored during incompatibility testing; however, the fourth return value, if supplied, should be an explanation of what caused the incompatibility, as a list. If none is returned, CLASSIC will generate an explanation for you; however, it may not be as instructive as you would like.

To see how we might use this let us return to our definition of **fills-substring**. We might have defined the function this way:

```
(defun fills-substring (ind role-name string)
    (let* ((r (cl-named-role role-name))
           (fillers (cl-fillers ind r))
           (ret (find-if
                  #'(lambda (filler)
                       (AND (stringp filler)
                            (case-insensitive-search
                                 string filler))
                  fillers))))
       (cond (ret t)
```

```
((cl-ind-closed-role? ind r)
 (values nil nil nil
     (list 'role-cannot-contain-string role-name string)))
(t '?))))
```

Now trying to define this concept:

```
CONCEPT16:  (AND (AT-LEAST 1 name)
                 (AT-MOST 1 name)
                 (TEST-C fills-substring name "Tom")
                 (FILLS name "Kirk")
                 ...)
```

will generate an error including this inference:

```
*CLASSIC ERROR*
...
(ROLE-CANNOT-CONTAIN-STRING NAME "Tom")
...
```

9.5 When and How to Use the Test Extensions

The mechanisms described so far are very powerful. It is possible to use them to extend the reasoning capability of CLASSIC almost arbitrarily. While this is an extremely useful ability, especially in real-world applications and embedded systems, it does come with a price. In particular, adding complex reasoning capabilities adds computational complexity, increasing execution time.

9.5.1 When to Use the Test Extensions

The extended functionality described here is included to support users who need additional expressive and inferential power and who are willing to pay a price for it. The extended functionality is distributed among many functions in order to make it easier to choose only that functionality that is useful. Therefore, before using the extension mechanisms, it is worthwhile spending time thinking specifically about how and when the extensions should be used for a particular application. This is an integral and important part of designing the structure and ontology of your domain. A carefully planned domain can greatly increase productivity. A poorly designed one can cause headaches and delays. The same is true for developing new operators via the test extension mechanisms. It is important to decide exactly what you want, how all of the operators interact with each other and the rest of CLASSIC and exactly how complete you need your system to be.

For example, doing date reasoning correctly and completely is a very difficult task. Further, the amount of reasoning necessary can require a great deal of time. On the other hand, for many applications, operators like `date-before`, `date-after` and `date-between` would be more than sufficient.

9.5.2 How to Use the Test Extensions

CLASSIC strives to be timely. In many situations, CLASSIC uses internal representations to speed up processing of other inferences such as subsumption or individual recognition. Similarly, one may wish to use an internal representation for test concepts.

For example, when comparing dates, it would be faster if they were encoded as single numbers rather than lists of three integers. Such a conversion might consist of a function that maps a date to the number of days from some reference point, or something simpler like a mapping from (y,m,d) to $(y - 1900) * 10000 + m * 100 + d$. In the second case, the normalization of `(date-after arrival "August 21 1995")` would be `(date-after arrival 950821)`. A similar representation would be used by **date-before** and **date-between**.

Thereafter, it is this normalized test expression which is passed as arguments to other functions. As a result all the test processing functions would proceed faster because normal LISP integer operators could be used. For example, let us return to the conjunction function **conjoin-date-betweens**, defined in Section 9.3:

```
(defun conjoin-date-betweens (nlist1 nlist2)
   (let* ((d11 (apply #'convert-to-date (subseq nlist1 2 5)))
          (d12 (apply #'convert-to-date (subseq nlist1 5 8)))
          (d21 (apply #'convert-to-date (subseq nlist2 2 5)))
          (d22 (apply #'convert-to-date (subseq nlist2 5 8)))
          (r1 (cl-named-role (cadr nlist1)))
          (r2 (cl-named-role (cadr nlist2)))
          (pr (pick-descendant r1 r2))
          (dt-overlap (get-date-overlap d11 d12 d21 d22)))
     (cond ((not pr) t)
           ((not dt-overlap)
            (values nil '(date-ranges-do-not-overlap)))
           (t (normalize-date-between
                  (append (list 'date-between (cl-role-name pr))
                          (date-to-numbers (car dt-overlap))
                          (date-to-numbers (cdr dt-overlap))))))))
```

With a representation maintaining dates as lists of numbers, a test expression's list must be converted each time it is used. In the representation where a date is a simple whole number, this is no longer necessary. In fact, if this is the internal representation of the `DATE-OBJECT`s stored in roles, the **date-between** function itself should be much faster.

There are trade-offs in this kind of decision. In this case, the internal representation is quicker and more compact; however, it is also more difficult for a human to understand. In particular, a user making use of the explanation system will have to deal with explanations involving expressions like `(date-between arrival 950101 960202)`. Even users who are not using explanation and are simply looking at objects will see such possibly confusing expressions.

The lesson here is simple. Trade-offs exist between readability and speed. It is up to the programmer to decide the value of these trade-offs.

9.6 Explanation

9.6.1 Using Explanation With Test Extensions

The explanation facility in CLASSIC (see Section 7) is useful in a variety of ways. It provides end users with the ability to understand the results of complicated inferences. It also provides advanced users and programmers with a nice debugging tool.

Because this is such an important facility, CLASSIC provides a mechanism for providing specific explanations for the results of reasoning with test extensions. If explanations are not provided by the programmer, CLASSIC will generate a default explanation. Usually these default explanations are sufficiently informative for a particular result.

For example, consider **conjoin-date-betweens**. Using it, CLASSIC would be able to simplify

```
(AND (date-between arrival 1995 1 1 1995 12 31)
     (date-between departure 2000 1 1 2000 12 31)
     (date-between arrival 1995 6 1 1996 12 31))
```

to

```
(AND (date-between arrival 1995 6 1 1995 12 31)
     (date-between departure 2000 1 1 2000 12 31))
```

Further, when asked why it deduced the simplified test expression, using `cl-exp`, CLASSIC would generate an explanation of the form:

```
TESTS-SIMPLIFY: (date-between arrival 1995 1 1 1995 12 31)
                (date-between arrival 1995 6 1 1996 12 31)
      →         (date-between arrival 1995 6 1 1995 12 31)
```

Similarly, after simplifying

```
(AND (date-between arrival 1995 1 1 1995 12 31)
     (date-between arrival 1995 12 1 2000 12 31)
     (date-between arrival 1995 6 1 1996 12 31))
```

to

```
(date-between arrival 1995 12 1 1995 12 31)
```

it would generate an explanation of the form:

```
TESTS-SIMPLIFY: (date-between arrival 1995 1 1 1995 12 31)
                (date-between arrival 1995 12 1 2000 12 31)
                (date-between arrival 1995 6 1 1996 12 31)
      →         (date-between arrival 1995 12 1 1995 12 31)
```

using as many of the original test expressions as necessary to generate the final result.

Having said all of this, it is still worth noting that sometimes the default explanations are not at all obvious to an end-user, even when they seem obvious to the implementor. Therefore, we recommend taking advantage of the opportunity to provide detailed explanations whenever possible.

To illustrate this, let us return to the **fills-substring-normalize** normalization function. This function generates very detailed errors. For example:

```
(cl-create-ind 'an-ind '(fills-substring name 3))
```

would generate an error including the inference:

```
*CLASSIC ERROR*
...
NOT-A-STRING: 3
...
```

Similarly:

```
(cl-create-ind 'another-ind (fills-substring nme "Tom"))
```

would generate an error including the inference:

```
*CLASSIC ERROR*
...
NOT-A-ROLE: NME
...
```

providing enough specific information that a user could notice that `name` had been misspelled as `nme`. Without such inferences provided by **fills-substring-normalize**, CLASSIC will only indicate that *some* error was made in the **fills-substring** expression. Even in a simple test function like **fills-substring**, it may take some time to determine the error without any other information, especially for a naive user. As the number of different kinds and sources of errors grows for a particular test function, the task becomes too burdensome.

Similarly, without specific inferences explaining normalization, simplification and consistency reasoning for tests, it can become very difficult for even an experienced CLASSIC user to follow a complex series of inferences and interactions. Again, whenever there is any chance that the source of an error or inference might be ambiguous it is worthwhile to provide a meaningful explanation.

9.6.2 Detailed Explanations With Subsumption Functions

A test subsumes another test whenever the first test is more general. That is, a test subsumes a second test whenever it is the case that if the second test is true of an individual then the first test must also be true of the individual. A set of tests subsumes another set of tests when it is the case that each of the tests in the first set subsumes at least one test in the second. For example, this concept:

```
CONCEPT17:  (AND (fills-substring name "Tom")
                 (fills-substring name "Kirk"))
```

subsumes this concept

```
CONCEPT18:  (AND (fills-substring name "Tommy")
                 (fills-substring name "stomp")
                 (fills-substring name "Tom Kirk")
                 (fills-substring name "Captain Kirk")
                 (fills-substring name "Spock"))
```

This is true because each test of CONCEPT17 subsumes *some* test of CONCEPT18. In particular (**fills-substring** name "Tom") subsumes the first three test expressions of CONCEPT18 and (**fills-substring** name "Kirk") subsumes the third and fourth test of CONCEPT18 (note that neither test subsumes the final test of CONCEPT18; however, that does not matter).

During explanation, it is the philosophy of CLASSIC to provide all the ways that a fact may have been inferred instead of just the first way it happened to have been inferred. This is usually desirable in explanation; however, explanations involving test extensions may not benefit from this view. For example, in the case above, explaining why CONCEPT17 subsumes CONCEPT18 would necessitate generating quite a few lines.

The amount of information to present about test subsumption during explanation is determined by the value of the pruning variable ***cl-exp-tests-level***. Its value can be 0, 1 or 2. A value of 0 (the default) means that CLASSIC should indicate only that the subsumer's tests subsume the subsumee's tests in some way. If they subsume because of textual equality then the inference TEST-SUBSET is used. If they do not subsume just because of textual equality then the inference TEST-SUBSET-BY-SUBSUMPTION is used. Explanation inferences are documented in Section 7.4.

A value of 1 means to show explicitly one subsumed test for each test in the subsuming concept. For every test in the subsuming concept, CLASSIC will find one test from the subsumee that it subsumes and generate an inference explaining the test subsumption. If there is an explanation returned by the test subsumption function, it is used; otherwise, the inference TEST-SUBSUMES is used.

A value of 2 means to show explicitly every subsumed test for each test in the subsuming concept. For every test in the subsuming concept, CLASSIC will find all the tests from the subsumee that it subsumes and, for each one, generate an inference explaining the test subsumption. If there is an explanation returned by the test subsumption function, it is used; otherwise, the inference TEST-SUBSUMES is used.

There is a way to ask for this explanation directly, using the function **cl-exp-tests-subsume?**. The syntax is

```
(cl-exp-tests-subsume?
      <derived concept> <CLASSIC object>
      &optional (verbose?  NIL))
```

This function takes a concept and a concept or individual. The tests of the first argument (the subsumer) are tested to see if they subsume the tests of the second argument (the subsumee). The function **cl-exp-tests-subsume?** returns a list of explanation inferences indicating that each of the subsumer's tests does indeed subsume some test in the subsumee. If there is not a match for every one of the subsumer's tests, `NIL` is returned.

If the optional argument `verbose?` is `NIL`, **cl-exp-tests-subsume?** will find only one corresponding test in the subsumee. If `verbose?` is non-`NIL` then **cl-exp-tests-subsume?** will find every corresponding test in the subsumee.

9.6.3 Detailed Explanations With Conjunction Functions

As mentioned before, it is the philosophy of CLASSIC to provide all possible ways that a fact may have been inferred instead of just the first way it happened to be inferred. For example, this expression:

```
(AND (date-between arrival 1995 1 1 1995 5 1)
     (date-between arrival 1995 2 1 1995 6 1)
     (date-between arrival 1995 2 1 1995 7 1)
```

simplifies to

```
(date-between arrival 1995 2 1 1995 5 1)
```

This final result could come from three combinations:

```
(AND (date-between arrival 1995 1 1 1995 5 1)
     (date-between arrival 1995 2 1 1995 6 1))
     → (date-between arrival 1995 2 1 1995 5 1)

(AND (date-between arrival 1995 1 1 1995 5 1)
     (date-between arrival 1995 2 1 1995 7 1))
     → (date-between arrival 1995 2 1 1995 5 1)

(AND (date-between arrival 1995 1 1 1995 5 1)
     (date-between arrival 1995 2 1 1995 6 1)
     (date-between arrival 1995 2 1 1995 7 1))
     → (date-between arrival 1995 2 1 1995 5 1)
```

CLASSIC will only generate one of the explanations for test simplification. In particular, it will be the first combination that happens to produce the final answer (the details of the algorithm are such that it would be the first combination listed above, but one cannot depend on that). The reason for this is simple. Determining all ways a result could be obtained requires testing all possible combinations of tests. This is an exponential operation and can take an incredible amount of time even for just a few tests. Because CLASSIC strives to be timely, it does not perform the search.

On the other hand, it may be useful to be able to determine such information. The function **cl-test-sets-producing** will provide all possible sources of a particular result. The syntax is

```
(cl-test-sets-producing
       <test expression> <list of test expressions>
       &key (skip-singles NIL) (skip-supersets NIL))
```

All combinations of the list of test expressions are conjoined in order to see which produce the test expression in the first argument. The test expression may be `NIL`. In this case, **cl-test-sets-producing** looks for incompatible tests. The result of this function is a list of lists. Each sublist is a list of tests that produce the desired result. For our example above,

```
(cl-test-sets-producing
       '(date-between arrival 1995 2 1 1995 5 1)
       '((date-between arrival 1995 1 1 1995 5 1)
         (date-between arrival 1995 2 1 1995 6 1)
         (date-between arrival 1995 2 1 1995 7 1)))
```

would produce

```
( ( (date-between arrival 1995 1 1 1995 5 1)
    (date-between arrival 1995 2 1 1995 6 1) )
  ( (date-between arrival 1995 1 1 1995 5 1)
    (date-between arrival 1995 2 1 1995 7 1) )
  ( (date-between arrival 1995 1 1 1995 5 1)
    (date-between arrival 1995 2 1 1995 6 1)
    (date-between arrival 1995 2 1 1995 7 1) ) )
```

The keywords `skip-singles` and `skip-supersets` prune the search space. If `skip-singles` is `T`, **cl-test-sets-producing** will not check to see if any of the test expressions in the list of test expressions is exactly the same as the desired result. If `skip-supersets` is `T` then only minimal combinations will be tested:

```
(cl-test-sets-producing
       '(date-between arrival 1995 2 1 1995 5 1)
       '((date-between arrival 1995 1 1 1995 5 1)
         (date-between arrival 1995 2 1 1995 6 1)
         (date-between arrival 1995 2 1 1995 7 1))
       :skip-supersets T)
```

would generate only the first two combinations because the third is a superset of them both:

```
( ( (date-between arrival 1995 1 1 1995 5 1)
    (date-between arrival 1995 2 1 1995 6 1) )
  ( (date-between arrival 1995 1 1 1995 5 1)
    (date-between arrival 1995 2 1 1995 7 1) ) )
```

It is wise to set `skip-supersets` to `T` whenever possible because it can dramatically speed up the searching process.

9.7 Upcoming Features

There are several features that will be added to those discussed in this section. A brief description of some of the more imminent ones is provided below.

9.7.1 Ordering

During conjunction, it is sometimes useful to know the order in which functions will be conjoined. In some cases, it is possible to optimize one's conjunction functions to take advantage of this order. In other cases, it makes more sense to try to conjoin the most restrictive tests first (because these are the most likely to generate inconsistencies).

Currently, CLASSIC chooses a canonical order for test expressions that determines the order in which they will be conjoined. Eventually, it will be possible to override this default and specify how tests should be ordered.

9.7.2 Printing

The printing extension will allow for a description of how particular tests should be printed.

9.7.3 Subsumption

Currently, the test extensions only allow for the description of subsumption relationships between test expressions. Eventually, it will be possible to describe subsumption relationships between tests and other CLASSIC operators. This way, CLASSIC can directly determine that (**date-between** `arrival 1995 9 1 1995 9 3`) subsumes (**FILLS** `arrival` *September 2 1995*).

9.7.4 Conjunction and Implication

As with subsumption, it is sometimes necessary to conjoin test expressions with standard CLASSIC operators, such as `all` restrictions. Moreover, in the most general case, a test expression or a regular CLASSIC description may be implied by the conjunction of one or more constructs, of similarly varied nature. To achieve full generality, future extensions will allow test expressions to interact with CLASSIC constructors.

9.7.5 Macros

For many functions normalization and subsumption can be easily described by the types of their arguments. For example, **fills-substring** can be described as a function that accepts the type `(role string)`. From this information, it is easy to generate a normalization function. Similarly, the subsumption relationship for **fills-substring** follows naturally from the relationships between the arguments. This relationship could be described as `(ancestor-of substring-of)`. As we receive more feedback about the uses of the test extension functions, we will provide these kinds of macros for typical cases.

APPENDICES

A The CLASSIC Grammar

The following defines the syntax for typing in a CLASSIC concept or individual description:

<descr> ::= THING | <**CLASSIC**-descr> | <**HOST**-descr>
<**HOST**-descr> ::= HOST-THING | <**HOST**-concept-name> |
(**AND** <**HOST**-descr>$^+$) |
(**ONE-OF** <**HOST**-individual>*) |
(**TEST-H** <fn> <arg>*) |
(**MIN** <number>) |
(**MAX** <number>)
<**CLASSIC**-descr> ::= CLASSIC-THING | <**CLASSIC**-concept-name> |
(**AND** <**CLASSIC**-descr>$^+$) |
(**ONE-OF** <**CLASSIC**-individual-name>*) |
(**TEST-C** <fn> <arg>*) |
(**ALL** <role-name> <descr>) |
(**AT-LEAST** <positive-integer> <role-name>) |
(**AT-MOST** <non-negative-integer> <role-name>) |
(**FILLS** <role-name> <individual-name>$^+$)
(**SAME-AS** (<attribute-name>$^+$) (<attribute-name>$^+$))
<**HOST**-concept-name> ::= <symbol> *;; defined as a host concept*
<**CLASSIC**-concept-name> ::= <symbol> *;; defined as a classic concept*
<role-name> ::= <mrole-name> | <attribute-name>
<mrole-name> ::= <symbol> *;; defined as a role, but not functional*
<attribute-name> ::= <symbol> *;; defined as a functional role*
<individual-name> ::= <**HOST**-individual> | <**CLASSIC**-individual-name>
<**HOST**-individual> ::= <string> | <number> |
'<CommonLisp-expr> | (quote <CommonLisp-expr>)
<**CLASSIC**-individual-name> ::= <symbol>
<fn> ::= a function in the **HOST** language (Common LISP)
that takes an individual, plus the extra arguments,
and returns T, ?, or NIL.
<arg> ::= an expression passed to the test functions.

Alternative Keyword

= may be used instead of the **SAME-AS** operator.

B The CLASSIC Reader

There is a special notation in the CLASSIC system for accessing a told or derived concept or individual, a role, or a rule with a particular name.

@`c`{name} or @`C`{name} will return the derived concept with the given name.

@`tc`{name} or @`TC`{name} will return the told concept with the given name.

@`i`{name} or @`I`{name} will return the derived **CLASSIC** individual with the given name.

@`ti`{name} or @`TI`{name} will return the told **CLASSIC** individual with the given name.

@`r`{name} or @`R`{name} will return the role with the given name.

@name will return the CLASSIC object (derived concept, derived **CLASSIC** individual, role, or rule) with the given name, as long as there is only one CLASSIC object with that name. Otherwise, it will signal an error.

In addition, when a CLASSIC object is returned, it is printed as @`c`{name}, @`tc`{name}, @`i`{name}, @`ti`{name}, or @`r`{name} for a derived concept, a told concept, a derived **CLASSIC** individual, a told **CLASSIC** individual, or a role, respectively. Note that when a CLASSIC rule is returned, it is printed as @`rule`{name}, but this notation cannot currently be used to access a rule—you must use @name (the rule must be the only CLASSIC object with that name) or one of the functions **cl-named-rule** or **cl-named-object** (see Appendix D.9.2.1).

C Some Definitions

We here define some terms you will need to know in order to understand the function descriptions in Appendix D.

The *parents* of a derived concept are the most specific classified derived concepts that subsume it, while the *ancestors* are all the classified derived concepts that subsume it. The *children* of a derived concept are the most general classified derived concepts that it subsumes, while the *descendants* of a derived concept are all the classified derived concepts that it subsumes. There are no ancestors of a derived concept that are children of its parents, i.e., there are no cycles.

The *parents*, or *direct parents* of a derived individual are the most specific classified derived concepts which the individual satisfies, and the individual is referred to as a *direct instance* of these concepts. The *ancestors* of a derived individual are all the classified derived concepts which the individual satisfies, and the individual is referred to as an *instance* of these concepts.

D CLASSIC Function and Variable Descriptions

The CLASSIC functions are grouped according to whether they are related to concepts, individuals, or roles, and then whether they are used to add new information to the network, retrieve information, or do predicate testing. In general, CLASSIC functions begin with the prefix "`cl-`".

D.1 General

D.1.1 Variables

classic-source-directory A pathname or string that specifies the directory where the CLASSIC source files are stored. Used by **cl-load-files** and **cl-compile-files**. Note: if your LISP sets the ***load-pathname*** variable to the path of the file being loaded (see Appendix I.4), then when the CLASSIC load file is loaded, ***classic-source-directory*** will, by default, refer to the directory containing this file.

classic-binary-directory A string which is the directory where the CLASSIC binary files are stored. Used by **cl-load-files** and **cl-compile-files**. Note: if your LISP sets the ***load-pathname*** variable to the path of the file being loaded (see Appendix I.4), then when the CLASSIC load file is loaded, ***classic-binary-directory*** will, by default, refer to the directory containing this file.

cl-error-object-list Points to a list of "error objects" from the last inconsistency error that was generated. These are copies of all the individuals that were modified as a result of the problematic input, and they contain information derived from any inferences performed based on this input. Even though each of these individuals has the same name as a real individual, it is not **eq** to the real individual (they are not the same LISP object). The function **cl-error-object** is used to access objects on this list. See Section 6 for a description of error handling, or Appendix D.1.4 for a description of **cl-error-object**.

D.1.2 Miscellaneous

(**cl**) Starts up the command line interface (see Appendix G for a description of the interface). When the interface is running, the prompt is **Classic>**. The **QUIT** command exits the interface.

(**cl-startup**) Initializes the knowledge base to contain just the built-in concepts—`THING`, `CLASSIC-THING`, `CONCEPT`, `HOST-THING`, and the built-in **HOST** concepts (see Section 2.1.5). Sets the debugging level to `0`, turns the warning mode on, and turns the error mode off (see **cl-set-classic-debug-level**, **cl-set-classic-error-mode**, and **cl-set-classic-warn-mode** below). See Appendix D.3.1 to see how to set up startup hook functions—functions that will get executed every time **cl-startup** is called.

(**cl-clear-kb**) Removes all individuals from the knowledge base, except those mentioned in **ONE-OF** and **FILLS** restrictions, as part of named concepts and rules. These individuals revert to `CLASSIC-THING`.

(**cl-reader-init**) Fixes up the read table for CLASSIC, so the @ macro will work correctly. This is not needed under normal circumstances, but may be needed if a LISP image containing CLASSIC is being used.

(**cl-run-compatibility-package** &optional[20] (run? `T`)) If run? is `T` (the default), turns the compatibility package on. Otherwise, turns the compatibility package off. When the compatibility package is on, any description from the description language of CLASSIC 1 should work in CLASSIC 2 (i.e., descriptions containing the **PRIMITIVE** and **DISJOINT-PRIMITIVE** operators). See the release notes for a description of the differences between CLASSIC 1 and CLASSIC 2.

(**cl-load-files** &key (:lisp? `NIL`)
(:user-dir ***classic-source-directory*** or ***classic-binary-directory***))[21] Loads all the CLASSIC files from the directory specified by :user-dir. If the :lisp? keyword has a non-`NIL` value (the default is `NIL`), then :user-dir defaults to ***classic-source-directory***, and the LISP files are loaded; otherwise, :user-dir defaults to ***classic-binary-directory***, and the binary files are loaded.

(**cl-compile-files** &key (:user-dir ***classic-source-directory***)
(:dest-dir ***classic-binary-directory***))
Compiles all the CLASSIC files. The :user-dir keyword, which defaults to the value of the variable ***classic-source-directory***, is the directory where the LISP files are stored. The :dest-dir keyword, which defaults to the value of the variable ***classic-binary-directory***, is the directory where the binary files are to be stored.

(**cl-print-global-settings**) Prints the global settings for CLASSIC.

D.1.3 Predicates

(**cl-type?** <object> <type>) Returns `T` if the object is of the given CLASSIC type (i.e., concept, told-concept, role, rule, (**CLASSIC**) individual or told-individual); otherwise `NIL`.

(**cl-classic?** <object>) Returns `T` if object is a **CLASSIC** object (i.e., a told or derived concept, role, rule, or **CLASSIC** told or derived individual); otherwise `NIL`.

D.1.4 Retrieval

(**cl-type** <object>) Returns the CLASSIC type of an object ((derived) concept, told-concept, (**CLASSIC**) individual, (**CLASSIC**) told-individual, role, or rule), or `NIL` if the object is not a CLASSIC object.

(**cl-named-object** <symbol>) Returns the CLASSIC object (derived concept, derived individual, role, or rule) with the given name, or `NIL`. It signals an error if there is more than one object with this name. Equivalent to @{`symbol`} (the @ notation is described in Appendix B).

[20]The &optional keyword, used in function headers, says that the following argument is optional. If the argument is a two-element pair surrounded by parentheses (), the first element is the argument type and the second element is its default value, used when no argument is given.

[21]The &key keyword says that the following arguments have the default values given in parentheses. If you wish to change the value of a keyword argument, you must give the keyword name (starting with :), followed by the value. For example, if you call (**cl-load-files**), the :lisp? keyword has the value `NIL`. However, if you call (**cl-load-files** `:lisp? T`), the :lisp? keyword has the value `T`.

(**cl-named-objects** <symbol>) Returns a list of the CLASSIC objects (derived concept, derived individual, role, or rule) with the given name, or `NIL`.

(**cl-error-object** <symbol> &optional <type> (<list> ***cl-error-object-list***)
Takes the name of an error object, and an optional object type (concept or individual) and list of error objects, and returns the error object in the list which has the given name and type, or it returns the first object in the list with the given name, if no type is specified. See Section 6 for a description of error handling, or Appendix D.1.1 for a description of ***cl-error-object-list***.

(**cl-name** <object>) Returns the canonical name (a symbol) of a CLASSIC object. If the object is an unclassified told or derived concept, it returns `NIL`. If the object is not a told or derived concept, told or derived **CLASSIC** individual, role, or rule, it returns the object.

(**cl-names** <object>) Returns a list of all the names (symbols) of a CLASSIC object. If the object is an unclassified told or derived concept, it returns `NIL`. If the object is not a told or derived concept, told or derived **CLASSIC** individual, role, or rule, it returns a list containing the object.

(**cl-comment** <object>) Takes a named concept, **CLASSIC** individual, role, or rule, and returns the comment (a string) associated with the object, or `NIL`.

(**cl-version**) Returns a number representing the CLASSIC version that is currently loaded, i.e., 2.1.

D.1.5 Addition

(**cl-set-comment** <object> <string>) Takes a named concept, **CLASSIC** individual, role, or rule, and a string. Sets the string as the comment of the object.

D.2 Printing

D.2.1 Variables

cl-error-msg-stream This variable defaults to the value of ***standard-output***, but can be set to any stream. When an error is signaled, the error message is printed to this stream.

cl-warn-msg-stream This variable defaults to the value of ***standard-output***, but can be set to any stream. The warning messages are printed to this stream (by **cl-print-warn**—see Appendix D.2.3).

cl-debug-msg-stream This variable defaults to the value of ***standard-output***, but can be set to any stream. The debugging messages are printed to this stream (by **cl-print-debug**—see Appendix D.2.3).

cl-null-stream This variable is set to the null stream. Any output sent to this stream will be discarded. To avoid printing the messages associated with one of the variables described above, set that variable to ***cl-null-stream***. Do *not* modify this variable.

classic-print-right-margin The value of this variable, which defaults to 80, is used as the screen width for the object printing functions (**cl-print-object**, etc.). There is a minimum width of 40, below which the margin cannot shrink.

D.2.2 Mode Setting and Retrieval Functions

(**cl-set-classic-error-mode** &optional (<boolean> `NIL`)) Takes an optional boolean argument (the default is `NIL`), and turns the CLASSIC error mode on for a non-`NIL` value or off for `NIL`. If the CLASSIC error mode is on, then when a CLASSIC error is signaled, the LISP debugger will automatically be entered; otherwise error information will be returned.

(**cl-classic-error-mode**) Returns `T` if the CLASSIC error mode is turned on; otherwise `NIL` (the default).

(**cl-set-classic-warn-mode** &optional (<boolean> `T`)) Takes an optional boolean argument (the default is `T`), and turns the CLASSIC warning mode on for a non-`NIL` value or off for `NIL`. When the warning mode is on, CLASSIC will warn the user of certain events. See also Section 6, and **cl-print-warn** in Appendix D.2.3 for more information on warnings.

(**cl-classic-warn-mode**) Returns `T` (the default) if the CLASSIC warning mode is turned on; otherwise `NIL`.

(**cl-set-classic-debug-level** &optional (<level> `1`)) Takes an optional level argument, which must be a non-negative integer (the default is `1`), and sets the debugging level to that value. A value of `0` means no debugging messages will be printed. See also Section 6, and **cl-print-debug** in Appendix D.2.3 for information on error handling and printing debugging messages.

(**cl-classic-debug-level**) Returns a non-negative integer—the current debugging level. Initially, and after **cl-startup** has been called, the level is `0`—no debugging messages are printed.

D.2.3 Printing Functions

(**cl-print-version-info**) Prints information relevant to the CLASSIC version that is currently loaded.

(**cl-print-debug** <level> <string> <arg>*)[22] Takes a level (a positive integer), a control string to print out, and some arguments to that string, in the same form as the `format` function. When the current debugging level is greater than or equal to the level argument, it prints out the format string to the steam that ***cl-debug-msg-stream*** is bound to (the default is ***standard-output***), prefixed by the string "**". This function always returns `T`. For example, the user's application might check a particular variable, `ind`, and if it has more than 500 fillers for the `friend` role, the application might call this function as

(**cl-print-debug** 2
"The individual ~a has over 500 fillers for the `friend` role" ind).

If the variable `ind` is bound to the individual `Mary` when this function is called, CLASSIC would print this message as

"** The individual @i{MARY} has over 500 fillers for the `friend` role."

The debugging level is initially `0`. See **cl-set-classic-debug-level** in Appendix D.2.2 for information on setting the debugging level. Note: if the level is not a positive integer, it will default to `1`.

(**cl-print-warn** <string> <arg>*) Takes a control string to print out, and some arguments to that string, in the same form as the `format` function. When the CLASSIC warning mode has been turned on (it is on by default, or can be turned on by calling **cl-set-classic-warn-mode** (see Appendix D.2.2)), it prints out the format string to the stream that ***cl-warn-msg-stream*** is bound to (the default is ***standard-output***), prefixed by the string "*WARNING*: ". Always returns `T`. For example, if the variable `con` is found to be a potentially inconsistent concept, the user's application might call this function as

(**cl-print-warn** "The concept ~a may be inconsistent" con).

If the variable `con` is bound to the concept `C1` when this function is called, CLASSIC would print this message as

"*WARNING*: The concept @c{C1} may be inconsistent."

The following is a description of the information printed for the functions **cl-print-object**, **cl-print-concept**, **cl-print-told-concept**, **cl-print-all-concepts**, **cl-print-all-told-concepts**, **cl-print-ind**, **cl-print-told-ind**, **cl-print-all-inds**, and **cl-print-all-told-inds**. For a particular object, the pieces of information (if they exist) are printed in the order presented:

- the decompositions it belongs to (for a disjoint primitive told concept);
- the primitive ancestors (for a derived object);
- the parents: for a derived object: the most specific classified derived concepts that subsume this object; for a told object: the told parents;

[22]Note the difference between functions which take as arguments <*type*>* (i.e., **cl-print-debug** takes <arg>*), and those that take <list of *types*> (i.e., cl-top-concepts in Appendix D.6.2 takes <list of concepts>). In the first case, there may be any number of arguments (or none), each of the specified type, while in the second case, there is exactly 1 argument which must be a list containing elements of the specified type.

- the ancestors (for a derived object): the other concepts that subsume this object;
- the children and descendants (for a derived concept), and the direct instances and instances (for a non-**HOST** derived concept);
- the **ONE-OF** restriction, if it exists (for a derived object, it is a list of derived individuals; for a told object, it is a list of lists of told individuals, i.e., the list of **ONE-OF** restrictions the user specified when creating the object);
- the test restrictions (for a told or derived object);
- the **SAME-AS** restrictions (for a told object);
- the interval (**MIN** and **MAX**) restrictions (for a told or derived **HOST** concept);
- the fillers and role restrictions (for a told or derived object):
  - If an **ALL** restriction is a classified concept (see Section 2.1.8), only the name of the concept will be printed; otherwise, the concept information will be recursively printed out. If the **ALL** restriction is printed as "Equivalent by SAME-AS to role-path: <list of attributes>", where <list of attributes> might be (@r{`best-friend`}), it means that it is at the tip of a path of a **SAME-AS** restriction, and is related by a **SAME-AS** restriction to the **ALL** restriction gotten by following the specified attribute path from the top-level object.
  - A `!` after a role name means the role is an attribute, and has an implicit **AT-MOST** restriction of `1`.
  - Information following a double arrow (⇒) is an **ALL** restriction, while information following a single arrow (→) is a list of the required fillers (for concepts), or current fillers (for individuals), and whether or not the role is closed (for told individuals)—this is signaled by a `#`. Note: when printing the told information for an individual, if a role has some derived fillers but no told fillers, and has been explicitly closed, the fillers for the told individual will print out as `()#`.
- the rules which have this concept or one of its ancestors as antecedent (for a top-level concept);
- the local rules: those which have this concept as antecedent (for a top-level concept).

All the following printing functions print to *standard-output*.

(**cl-print-object** <CLASSIC-object>) Prints a CLASSIC object, and returns it. If the object is not a told or derived concept, a told or derived **CLASSIC** individual, a role, or a rule, it assumes it is a **HOST** individual, and prints out its parents. Note: this function no longer accepts :told and :derived keywords.

(**cl-print-concept** <concept> &key (:told `NIL`) :all) Prints a derived concept, and returns it. If :told is non-`NIL`, it also prints the told information for the concept. If the concept is primitive or disjoint primitive, the concept name is surrounded with *'s.

By default, the derived parents, ancestors, children, descendants, direct instances, and instances will not be printed for unclassified concepts (including temporary concepts), since this information is not stored and needs to be calculated. However, this can be overridden by calling the function with the keyword :all set to `T`. (You can *avoid* printing out this information for classified concepts by calling the function with the keyword :all set to `NIL`.)

(**cl-print-told-concept** <told concept>) Prints a told concept, and returns it. If the concept is primitive or disjoint primitive, the concept name is surrounded with *'s.

(**cl-print-all-concepts** &key (:told `NIL`) (:built-in `NIL`)) Prints all the derived, classified concepts in the knowledge base, and returns `T`. If :told is non-`NIL`, it also prints the told information for each concept. If :built-in is `NIL` (the default), it ignores all the built-in concepts (i.e., it doesn't print `CLASSIC-THING`, `INTEGER`, etc.).

(**cl-print-all-told-concepts** &key (:built-in `NIL`)) Prints all the told, classified concepts in the knowledge base, and returns `T`. If :built-in is `NIL` (the default), it ignores all the built-in concepts (i.e., it doesn't print `CLASSIC-THING`, `INTEGER`, etc.).

(**cl-print-ind** <individual> &key (:told `NIL`)) Prints a derived individual, and returns it. If :told is non-`NIL`, it also prints the told information for the individual (:told is ignored for **HOST** individuals).

(**cl-print-told-ind** <told individual>) Prints a told individual, and returns it.

(**cl-print-all-inds** &key (:told `NIL`)) Prints all the derived **CLASSIC** individuals in the knowledge base, and returns `T`. If :told is non-`NIL`, it also prints the told information for each individual.

(**cl-print-all-told-inds**) Prints all the told **CLASSIC** individuals in the knowledge base, and returns `T`.

(**cl-print-role** <role>) Prints a role (the name, ! for an attribute, the parent, children, and inverse), and returns it.

(**cl-print-rule** <rule> &key (:derived `T`) (:told `NIL`)) Prints a (simple, description, or filler) rule (the name and antecedent, and when applicable, the filter, consequent, function, args, and/or role), and returns it. The derived and/or told versions of the filter and consequent will be printed, depending on the value of the :derived and :told keywords.

D.3 Hooks into Classic

This section is for advanced users of CLASSIC. It is not necessary for a novice user to understand this section.

D.3.1 Error Hook and Startup Hook Functions

The user may define startup hook functions and error hook functions. All the startup hook functions (if there are any) get called every time **cl-startup** is called (after **cl-startup** reinitializes the knowledge base). These functions should take no arguments. (See Section D.1.2 for a description of **cl-startup**.) All the error hook functions (if there are any) get called every time an error is generated. The arguments to these functions should be as follows: 1) the error code; 2) the arguments to the error (if any); 3) the error role-path; and 4) the error object (the error role-path and error object may be `NIL` if this is not a "real" error). (See Section 6 for a description of error handling.)

(**cl-error-hooks**) Returns a list of all the user-defined error hook functions.

(**cl-add-error-hook** <function>) Adds a new error hook function to the list of error hook functions.

(**cl-remove-error-hook** <function>) Removes an error hook function from the list of error hook functions.

(**cl-startup-hooks**) Returns a list of all the user-defined startup hook functions.

(**cl-add-startup-hook** <function>) Adds a new startup hook function to the list of startup hook functions.

(**cl-remove-startup-hook** <function>) Removes a startup hook function from the list of startup hook functions.

D.3.2 Other Hooks into the Internals of Classic

There are a number of hooks into the internals of CLASSIC. Each hook has a variable associated with it, and each variable may be bound to a list of functions that are to be executed when the appropriate activity takes place in CLASSIC. There is a predefined hook function for each hook variable, and the hook variables can each be set to a list containing the appropriate predefined function in order to do some tracing of CLASSIC. The CLASSIC file **util.lisp** contains these predefined hook functions, which you can use as templates for writing your own hook functions. By default, each hook variable is set to `NIL`.

You can use the function **cl-hook-print** to write your own hook functions. This function takes a stream, a format string to print, and optional arguments to the format string. It prints *CL TRACE*: followed by the appropriate message. The syntax is

(**cl-hook-print** <stream> <msg> <args>*)

For examples of using this function, see the predefined hook functions in **util.lisp**.

The variable ***cl-hook-stream***, which defaults to the value of ***standard-output***, is the stream to which the predefined hook functions print. By resetting this variable, you can print to a different stream.

The function **cl-set-hook-defaults** takes an optional argument which, when set to `T` (the default), causes all the hook variables to be set to their predefined hook functions. When the argument is `NIL`, all the hook variables are set to `NIL`. The syntax is

(**cl-set-hook-defaults** &optional (<boolean> `T`))

The following is a simple example where you might want to write your own hook functions. Suppose you want a list of all individuals that change in any of the following ways: information is explicitly added or implicitly propagated to the individual, the individual is reclassified, or a rule is fired on the individual. In that case, the hook functions for the appropriate hooks (***cl-ind-add-hook***, ***cl-propagation-hook***, ***cl-inverse-role-propagation-hook***, ***cl-classify-ind-hook***, and ***cl-fire-rule-hook***) would just do a **pushnew** to a single list. You would then need to reset the list within your code, after each call to a user function that adds information to an individual, creates a rule, etc. Resetting the list is important, because otherwise: (1) the list could grow too large, and the **pushnew** calls would then be slow; and (2) the list would have extra information on it that would no longer apply.

A user might also want to use the information to do graphical animation, in which case, each hook function would immediately update a display.

For each hook, the following gives the name of the hook variable, the required arguments to its hook functions (in [ ]'s), the predefined hook function that may be used for that hook variable (in ( )'s), and a description of the hook.

cl-create-ind-hook [(1) the newly-created individual]; (**cl-create-ind-hook-default**). This hook is run when an individual is created, either explicitly with **cl-create-ind**, or implicitly, by appearing in a **ONE-OF** or **FILLS** expression.

cl-ind-add-hook [(1) the individual]; (**cl-ind-add-hook-default**). This hook is run when information is added to an individual by a user function, or when a retraction fails and information is added back to the individual. *Note: if* **cl-create-ind** *is called, but the individual already exists, this hook will be run rather than* ***cl-create-ind-hook***.

cl-classify-ind-hook [(1) individual; (2) old parents; (3) new parents]; (**cl-classify-ind-hook-default**). This hook is run when an individual is classified and is found to belong to new parent concepts that it didn't belong to previously. For example, suppose `Mary`'s `parent` is `HEALTHY-THING`, and we add the following information to `Mary`: `(`**ALL** `food PLANT)`. She now gets classified under the new parent concept `VEGETARIAN`, in addition to the previous parent concept `HEALTHY-THING`. The args for this example would be: `(Mary (HEALTHY-THING) (VEGETARIAN HEALTHY-THING))`. *Note: if a new concept is created, and a particular individual gets classified under that concept, this hook will not get called, but instead, the following hook will get called.*

cl-classify-concept-hook [(1) the newly-created concept]; (**cl-classify-concept-hook-default**). This hook is run when a new concept is created successfully. It is not called if there is an error during concept creation. *Notes: (1) the parent concepts, children concepts, and direct instances can be retrieved from the concept; (2) if the user is updating a graphical display, then she will need to remove from the direct instances of the parent concepts any direct instances of this new concept. For example, if* `Mary` *was previously classified as a* `VEGETARIAN`*, and we define the concept* `HEALTHY-VEGETARIAN`*, if* `Mary` *is now classified under* `HEALTHY-VEGETARIAN`*, she is no longer a direct instance of* `VEGETARIAN`.

cl-propagation-hook [(1) individual propagated to; (2) role-path; (3) information propagated; (4) individual propagated from]; (**cl-propagation-hook-default**). This hook is run when information is propagated from one individual to another, when an individual has an **ALL** restriction that gets propagated to its fillers, e.g., `Mary` has the restriction:

`(`**AND** `(`**ALL** `child ATHLETE) (`**FILLS** `child Bob)).`

The concept `ATHLETE` gets propagated to the individual `Bob`. The args for this example would be: (`Bob (child) ATHLETE Mary`). *Note: the second argument is a role-path, rather than a role, because information could be propagated along a path of roles, e.g.,* `(`**ALL** `child (`**ALL** `child ATHLETE)).`

cl-inverse-role-propagation-hook [(1) individual propagated to; (2) role; (3) filler]; (**cl-inverse-role-propagation-hook-default**). This hook is run when an individual `I1` has a role `r1` filled by another individual `I2`, and `r1` has an inverse `r2`. This causes `I2`'s `r2` role to be filled by `I1`. For example, if we tell CLASSIC that the roles `husband` and `wife` are inverses, and that `Mary` has the restriction: `(`**FILLS** `husband Bob)`, then CLASSIC will propagate the information onto `Bob`: `(`**FILLS** `wife Mary)`. The args for this example would be: (`Bob wife Mary`).

cl-fire-rule-hook [(1) individual; (2) rule]; (**cl-fire-rule-hook-default**). This hook is run when a rule fires on an individual, which causes new information to get added to that individual. For example, if there is a rule, `HEALTHY-VEG-RULE`, which says that if someone is a `VEGETARIAN`, then she is a `HEALTHY-THING`, and `Mary` is found to be a `VEGETARIAN`, the rule would fire, and `Mary` would then be a `HEALTHY-THING`. The args for this example would be: (`Mary HEALTHY-VEG-RULE`).

cl-add-rule-hook [(1) the rule]; (**cl-add-rule-hook-default**). This hook is run when a rule is created, currently before the rule is fired on the individuals. It is also called when there is an error during rule removal, and the rule is added back.

cl-remove-rule-hook [(1) the rule]; (**cl-remove-rule-hook-default**). This hook is run when a rule is removed, currently before the rule's information is removed from the individuals. It is also called when there is an error during rule creation.

cl-ind-remove-hook [(1) the individual]; (**cl-ind-remove-hook-default**). This hook is run when information is removed from an individual, either by an explicit retraction, or on any error involving an individual (addition, retraction, rule creation, deletion). On an error or a retraction, everything is removed from the affected individual, then this hook is run, and then the appropriate information is added back in. Thus, while the hook function is running, the individual will temporarily be empty. After this hook has been run, the appropriate hook functions are called (classify-ind, propagation hooks, rule-firing).

cl-uncreate-ind-hook [(1) the individual]; (**cl-uncreate-ind-hook-default**). This is called after an inconsistency has been detected during the creation of an individual, in which case the individual gets deleted. *Note: this hook is currently only run for a top-level individual creation, because individuals created from* **FILLS** *and* **ONE-OF** *expressions are not deleted.*

The error hook from the previous section has a predefined function, **cl-error-hook-default**, which is set from within **cl-set-hook-default**.

D.4 Dumping Facility

The dumping facility takes an existing CLASSIC knowledge base (which has been loaded into CLASSIC), and dumps it to a file in CLASSIC format. All the information which the user has told CLASSIC is included, so the knowledge base can be recreated by loading this new file.

The function **cl-dump-kb** takes a file, and dumps the told information from the loaded knowledge base into the file. The syntax is

(**cl-dump-kb** <file>), e.g. (**cl-dump-kb** "kb.lisp").

To reload the knowledge base into CLASSIC, just load the file which was created.

Note: for rule and test functions, you should define the named functions in a separate file, and then when you reload a dumped knowledge base, you should first reload the function definitions. Do not use unnamed lambda expressions for your functions, because CLASSIC will not dump them correctly. Also, for the arguments to rule and test functions, you should not pass in unprintable data structures. Named CLASSIC objects (e.g., concepts) may or may not print out correctly, depending on the Lisp you are using. The safest types of arguments are static things that will not change their value (e.g., symbols, strings, and numbers). You can pass in the names of CLASSIC objects and have the rule and test functions calculate the objects.

The dumping facility saves all the CLASSIC objects, but you will need to reset anything affecting the environment (e.g., hooks, debugging environment, etc.).

If you are loading a knowledge base that has been "dumped" using **cl-dump-kb**, and that knowledge base contains meta individuals, please load the file as follows:

(**cl-with-new-concepts-not-interesting** (load "dumped-file.lisp"))

In this way, automatic creation of meta individuals upon concept creation is suppressed.

D.5 Retrieval Functions for both Concepts and Individuals

The following retrieval functions are for both concepts and individuals. *Note: whenever a function is supposed to return a list of objects, i.e., a list of test restrictions, and there are no such restrictions, it returns* `NIL` *(the empty set) unless otherwise specified in the function description.*

D.5.1 Subsumption Retrieval Functions

(**cl-object-subsumes** <derived concept or individual> <role-path>
&optional (<type> :instances))

Takes a concept or an individual, a role-path, and an optional type argument, which defaults to `:instances`. It follows the role-path to an **ALL** restriction (if the role-path is `NIL`, just gets the derived concept or individual which was the first argument. If this object is an individual, it constructs a concept which is equivalent to a description of the individual). Takes this concept, and if type is `:instances`, retrieves all instances;

if type is `:children`, retrieves all children concepts; otherwise, returns this concept (which may be unnamed).

D.5.2 Retrieval Functions for Derived Objects

(**cl-one-of** <derived concept or individual>) Returns the derived **ONE-OF** restriction on the object (a list of either derived individuals or **HOST** individuals), or `T` for no restriction.

(**cl-tests** <derived concept or individual>) Returns the derived test restrictions on the object, where a test restriction is a list containing the test function followed by any arguments: (<fn> <arg>*), i.e., for the `SUCCESSFUL-PARTY` concept from Section 2.1.4.1, it would return `((`**same-number-fillers** `male-guests female-guests))`, where `male-guests` and `female-guests` are the names of two roles, and they are the two arguments to the **same-number-fillers** function.

(**cl-primitives** <derived concept or individual>) Returns the primitive ancestors of the object (a list of derived concepts). *Note: to retrieve the primitive parents, the user can call* **cl-bottom-concepts** *on the ancestors.*

(**cl-role-restrs** <derived concept or individual>) Returns a list of the roles restricted by the derived object.

(**cl-at-least** <derived concept or individual> <role>) Returns the derived **AT-LEAST** restriction (a non-negative integer) on the given object and role.

(**cl-at-most** <derived concept or individual> <role>) Returns the derived **AT-MOST** restriction (a non-negative integer) on the given object and role, or `T` for no restriction.

(**cl-all** <derived concept or individual> <role>) Returns the derived **ALL** restriction (a derived concept) on the given object and role.

(**cl-fillers** <derived concept or individual> <role>) Returns the derived fillers (a list of either derived individuals or **HOST** individuals) on the given object and role.

(**cl-same-as** <derived concept or individual>) Returns a list of all the **SAME-AS** restrictions on the given object, where a **SAME-AS** restriction is a list of two attribute lists. For example, if there is a single **SAME-AS** restriction on an object, equating someone's mother's husband with his father, the function would return `(((@r{mother} @r{husband}) (@r{father})))`.

D.5.3 Retrieval Functions for Told Objects

(**cl-told-one-ofs** <told concept or individual>) Returns a list of the told **ONE-OF** restrictions on the object, where each restriction is a list of either told individuals or **HOST** individuals.

(**cl-told-tests** <told concept or individual>) Returns a list of the told test restrictions on the object.

(**cl-told-role-restrs** <told concept or individual>) Returns a list of the roles restricted by the told object.

(**cl-told-at-leasts** <told concept or individual> <role>) Returns the told **AT-LEAST** restrictions (a list of positive integers) on the given object and role.

(**cl-told-at-mosts** <told concept or individual> <role>) Returns the told **AT-MOST** restrictions (a list of non-negative integers) on the given object and role, or `T` for no restriction.

(**cl-told-all** <told concept or individual> <role>) Returns the told **ALL** restriction (a told concept) on the given object and role.

(**cl-told-fillers** <told concept or individual> <role>) Returns the fillers (a list of either told individuals or **HOST** individuals) on the given object and role.

(**cl-told-same-as** <told concept or individual>) Returns a list of all the **SAME-AS** restrictions on the given object, where a **SAME-AS** restriction is a list of two attribute lists.

D.6 Concepts

D.6.1 Predicates

(**cl-concept?** <object>) Returns `T` if object is a derived concept; otherwise `NIL`.

(**cl-told-concept?** <object>) Returns `T` if object is a told concept; otherwise `NIL`.

(**cl-classic-concept?** <object>) Returns `T` if object is a derived **CLASSIC** concept; otherwise `NIL`.

(**cl-host-concept?** <object>) Returns `T` if object is a derived **HOST** concept; otherwise `NIL`.

(**cl-classic-told-concept?** <object>) Returns `T` if object is a told **CLASSIC** concept; otherwise `NIL`.

(**cl-host-told-concept?** <object>) Returns `T` if object is a told **HOST** concept; otherwise `NIL`.

(**cl-disjoint?** <derived concept><derived concept> &optional (<boolean> `NIL`)) Returns `T` if the two concepts are disjoint (i.e., they produce an inconsistent concept when **AND**'ed together); otherwise `NIL`. When the optional boolean argument is `T` (it defaults to **NIL**), then the function takes into account all the simple filterless rules associated with the concepts when determining compatibility.

(**cl-disjoint-list?** <list of derived concepts> &optional (<boolean> `NIL`)) Returns `T` if any of the concepts in the list are disjoint; otherwise `NIL`. When the optional boolean argument is `T` (it defaults to **NIL**), then the function takes into account all the simple filterless rules associated with the concepts when determining compatibility.

(**cl-concept-primitive?** <concept>) Returns `T` if the concept is primitive or disjoint primitive; otherwise `NIL`.

(**cl-concept-classified?** <derived concept>) Returns `T` if the concept is classified; otherwise `NIL` (see Section 2.1.8).

(**cl-told-concept-classified?** <told concept>) Returns `T` if the concept is classified; otherwise `NIL` (see Section 2.1.8).

(**cl-subsumes?** <derived concept><derived concept>) Returns `T` if the first concept subsumes (is more general than) the second concept; otherwise `NIL`. See also **cl-instance-of-expr?** in Appendix D.7.1.

(**cl-concept-equal?** <derived concept> <derived concept>) Returns `T` if each concept subsumes the other; otherwise `NIL`.

D.6.2 Retrieval

Note: see also Section D.5 for retrieval functions which operate on both concepts and individuals.

D.6.2.1 Converting between told and derived concepts

(**cl-told-concept** <derived concept>) Returns the canonical told concept associated with a derived concept. If passed an unclassified derived concept, returns `NIL` and prints a warning (if warnings are turned on).

(**cl-told-concepts** <derived concept>) Returns all the told concepts associated with a derived concept. If passed an unclassified derived concept, returns `NIL` and prints a warning (if warnings are turned on).

(**cl-derived-concept** <told concept>) Returns the derived concept associated with a told concept. If passed an unclassified told concept, returns `NIL` and prints a warning (if warnings are turned on).

D.6.2.2 Converting between names and concepts

(**cl-named-concept** <symbol>) Returns the derived concept with the given name, or `NIL`. Equivalent to @c{`symbol`} (the @c notation is described in Appendix B).

(**cl-named-told-concept** <symbol>) Returns the told concept with the given name, or `NIL`. Equivalent to @tc{`symbol`} (the @tc notation is described in Appendix B).

(**cl-concept-name** <derived concept>) Returns the canonical name of the concept, or `NIL` if the concept is unnamed.

(**cl-concept-names** <derived concept>) Returns a list of all the names of the concept, or `NIL` if the concept is unnamed.

(**cl-told-concept-name** <told concept>) Returns the canonical name of the told concept, or `NIL` if the concept is unnamed.

(**cl-told-concept-names** <told concept>) Returns a list of all the names of the told concept, or `NIL` if the concept is unnamed.

D.6.2.3 Converting between Meta Individuals and Derived Concepts

(**cl-concept-meta-ind** <derived concept>) Returns the existing meta individual associated with the concept, or creates a new one and returns it. See Section 2.4.

(**cl-concept-meta-ind-concept** <derived individual>) Returns the derived concept for which this individual is the meta individual.

D.6.2.4 Retrieval Functions for Derived Concepts

(**cl-concept-parents** <derived concept>) Returns a list of the most specific classified derived concepts which subsume this concept.

(**cl-concept-ancestors** <derived concept>) Returns a list of all the classified derived concepts which subsume this concept.

(**cl-concept-children** <derived concept>) Returns a list of the most general classified derived concepts that are subsumed by this concept.

(**cl-concept-descendants** <derived concept>) Returns a list of all the classified derived concepts that are subsumed by this concept.

(**cl-concept-direct-instances** <derived concept>) Returns a list of all the individuals which satisfy (are instances of) this concept and don't satisfy any of its children (sub-concepts). Note: a **HOST** concept is illegal here, since it might have an infinite number of instances.

(**cl-concept-instances** <derived concept>) Returns a list of all the individuals which satisfy this concept. Note: a **HOST** concept is illegal here, since it could have an infinite number of instances.

(**cl-top-concepts** <list of derived concepts>) Returns the concepts from the list not subsumed by any other concepts in the list.

(**cl-bottom-concepts** <list of derived concepts>) Returns the concepts from the list that do not subsume any other concepts in the list.

(**cl-concept-rules** <classified derived concept>) Returns a list of all rules which would fire on any individual that is classified under this concept. This includes rules which have this concept as antecedent and rules which have an ancestor of this concept as antecedent.

(**cl-concept-local-rules** <classified derived concept>) Returns a list of the told rules that have this concept as antecedent.

(**cl-concept-min** <derived concept>) Returns a number, which is the min restriction on the concept (for an interval), or `NIL` for no restriction.

(**cl-concept-max** <derived concept>) Returns a number, which is the max restriction on the concept (for an interval), or `NIL` for no restriction.

(**cl-concept-expr** <derived concept> &optional <print-length>) Returns a defining description for the concept. Takes an optional argument, which must be a positive integer. If this argument is specified, any **AND** expression within the description is printed only up to the specified length, followed by "...".

(**cl-concept-find-compatible** <derived concept> &optional (<derived concept> `THING`)) Returns all concepts that are equal to or descendants of the prune concept (the optional argument, which defaults to `THING`), that are not disjoint from the given concept (i.e., one individual could be an instance of the given concept and one of the returned concepts). Takes into account compatibility with simple filterless rules.

(**cl-concept-find-not-compatible** <concept> &optional (<concept> `THING`)) Returns all concepts that are equal to or descendants of the prune concept (the optional argument, which defaults to `THING`), that are disjoint from the given concept. Takes into account compatibility with simple filterless rules.

D.6.2.5 Retrieval Functions for Told Concepts

(**cl-told-concept-parents** <told concept>) Returns a list of the told parents of the concept.

(**cl-told-concept-primitive** <told concept>) If the concept is a regular primitive concept, returns the symbol `REGULAR-PRIMITIVE`; if it is a disjoint primitive concept, returns a list of symbols which are the decomposition names; otherwise (it is a defined concept) returns `NIL`. (Note: it returns `NIL` for an unclassified concept, since it could not be primitive.)

(**cl-told-concept-mins** <told concept>) Returns a list of numbers, which are the told min restrictions on the concept (for an interval).

(**cl-told-concept-maxes** <told concept>) Returns a list of numbers, which are the told max restrictions on the concept (for an interval).

(**cl-told-concept-expr** <told concept> &optional <print-length>) Returns a defining description for the concept. Takes an optional argument, which must be a positive integer. If this argument is specified, any **AND** expression within the description is printed only up to the specified length, followed by "...".

D.6.3 Creation/Modification

D.6.3.1 Creation

(**cl-define-concept** <symbol> <descr>) Takes a name and a description (the concept definition), and creates and classifies the concept. If the new concept is found to be identical to an existing classified concept, then the symbol is added as a new name for the existing concept. If an inconsistency is found, then the concept and its name are not added. If everything is successful, the new derived concept is returned. See Appendix A for the syntax of a CLASSIC description.

(**cl-define-primitive-concept** <symbol> <**CLASSIC**-descr>) Defines a primitive concept with the given name and description, and returns the new derived concept.

(**cl-define-disjoint-primitive-concept** <symbol> <**CLASSIC**-descr> <decomposition symbol or list of decomposition symbols>) Defines a disjoint primitive concept with the given name and description, and belonging to the given decomposition(s), and returns the new derived concept.

(**cl-parse-expression** <descr>) Takes a CLASSIC description and returns a temporary, unclassified told concept (see Sections 2.1.7 and 2.1.8).

(**cl-normalize-concept** <unclassified told concept>) Takes an unclassified told concept and returns a temporary (unclassified) derived concept. If passed a classified told concept, will return the corresponding derived concept.

(**cl-create-ind-descriptor** <derived **CLASSIC** individual>) Creates a temporary derived concept which contains all the derived information in the individual (including information which came from rule-firings).

D.6.3.2 Modification

Once a concept has been defined, its definition cannot be modified, and it cannot be deleted. However, rules with the concept as antecedent can be added and removed—see Appendix D.9.3.

D.6.3.3 Name Modification

(**cl-define-concept-synonym** <symbol> <classified told concept>) Takes a symbol (a new name) and an existing told concept. Gives the told concept a synonym (the new name). The told concept will still print out as @tc{<old name>}, but @tc{<new name>} can be used to retrieve the concept.

(**cl-concept-change-canonical-name** <symbol> <classified told concept>) Takes a symbol (a name) and an existing told concept. Changes the canonical name of the told concept to the given name, and leaves all existing names. The name may or may not be an existing name for the concept. Note: if the told concept is not the canonical one, this function will not change the canonical name of the derived concept. The told concept will now print out as @tc{<new canonical name>}, but @tc{<old canonical name>} can still be used to retrieve the concept.

(**cl-remove-concept-names** <classified told concept> &optional (<list of symbols> `NIL`)) Removes the given list of names, or all the names (when <list of symbols> is `NIL`), as names for the concept, and returns the told concept. It will print a warning if any of the names in the list are not really names for the concept. If the list of names includes the canonical name, and there are names left for the concept, there is no way for the user to know which of the remaining names will become the canonical one. It is illegal to remove the canonical name of a built-in concept.

D.7 Individuals

D.7.1 Predicates

(**cl-ind?** <object>) Returns `T` if the object is a derived **CLASSIC** individual or a **HOST** individual (not a told or derived concept, a role, or a told **CLASSIC** individual); otherwise `NIL`.

(**cl-told-ind?** <object>) Returns `T` if the object is a told **CLASSIC** individual or a **HOST** individual (not a told or derived concept, a role, or a derived **CLASSIC** individual); otherwise `NIL`.

(**cl-classic-ind?** <object>) Returns `T` if the object is a derived **CLASSIC** individual; otherwise `NIL`.

(**cl-told-classic-ind?** <object>) Returns `T` if the object is a told **CLASSIC** individual; otherwise `NIL`.

(**cl-instance?** <derived individual><derived concept>) Determines whether or not the individual is an instance of (satisfies) the concept.

(**cl-instance-of-expr?** <object> <expression>) If object is a concept, returns `T` if the concept is subsumed by the concept defined by expression and `NIL` otherwise. If object is an individual, returns `T` if the individual is an instance of the concept defined by expression and `NIL` otherwise.

(**cl-ind-equal?** <derived individual> <derived individual>) Returns `T` only if both individuals are the same individual.

(**cl-ind-closed-role?** <derived **CLASSIC** individual> <role>) Returns `T` if the role on the given individual can have no more fillers (either because it has been explicitly closed, or because the number of fillers equals the **AT-MOST** restriction); otherwise `NIL`.

(**cl-told-ind-closed-role?** <told **CLASSIC** individual> <role>) Returns `T` if the role has been closed by the user on the given individual (with **cl-ind-close-role**); otherwise `NIL`.

(**cl-ind-role-value?** <derived **CLASSIC** individual> <role> <value>) Returns `T` if the role has the specified value for the individual; otherwise `NIL`.

D.7.2 Retrieval

Note: see also Section D.5 for retrieval functions which operate on both concepts and individuals.

D.7.2.1 Converting between told and derived individuals

(**cl-told-ind** <derived individual>) Returns the told individual associated with a derived individual (for a host individual, this is just the individual).

(**cl-derived-ind** <told individual>) Returns the derived individual associated with a told individual (for a host individual, this is just the individual).

D.7.2.2 Converting between names and individuals

(**cl-named-ind** <symbol>) Returns the derived **CLASSIC** individual with the given name, or `NIL`. Equivalent to @i{`symbol`} (the @i notation is described in Appendix B).

(**cl-named-told-ind** <symbol>) Returns the told **CLASSIC** individual with the given name, or `NIL`. Equivalent to @ti{`symbol`} (the @ti notation is described in Appendix B).

(**cl-ind-name** <derived individual>) If a **CLASSIC** individual, returns the canonical name of the individual. Otherwise (for a **HOST** individual), returns the individual.

(**cl-ind-names** <derived individual>) If a **CLASSIC** individual, returns a list of all the names of the individual. Otherwise (for a **HOST** individual), returns a list containing the individual.

(**cl-told-ind-name** <told individual>) If a **CLASSIC** individual, returns the canonical name of the individual. Otherwise (for a **HOST** individual), returns the individual.

(**cl-told-ind-names** <told individual>) If a **CLASSIC** individual, returns a list of all the names of the individual. Otherwise (for a **HOST** individual), returns a list containing the individual.

D.7.2.3 Retrieval Functions for Derived Individuals

(**cl-ind-parents** <derived individual>) Returns a list of the most specific classified derived concepts which the individual satisfies.

(**cl-ind-ancestors** <derived individual>) Returns a list of all the classified derived concepts which the individual satisfies.

(**cl-ind-rules** <derived individual>) Returns a list of all the rules which have any of the parents of the individual as antecedent. (Note: not all these rules may have fired on the individual, because for the purpose of this function, the individual is not tested for satisfying the filters.)

(**cl-ind-expr** <derived **CLASSIC** individual> &optional <print-length>) Returns a defining description for the individual. Takes an optional argument, which must be a positive integer. If this argument is specified, any **AND** expression within the description is printed only up to the specified length, followed by "...".

D.7.2.4 Retrieval Functions for Told Individuals

(**cl-told-ind-parents** <told **CLASSIC** individual>) Returns a list of the parents of the told individual (classified told concepts).

(**cl-told-ind-expr** <told **CLASSIC** individual> &optional <print-length>) Returns a defining description for the individual. Takes an optional argument, which must be a positive integer. If this argument is specified, any **AND** expression within the description is printed only up to the specified length, followed by "...".

D.7.3 Addition

(**cl-create-ind** <symbol><**CLASSIC**-descr>) If there is an existing **CLASSIC** individual with the given name, adds the information in the description to the individual; otherwise, creates a new **CLASSIC** individual with the given information. The derived individual is then classified and returned. See Appendix A for the syntax of a **CLASSIC** description.

(**cl-ind-add** <derived **CLASSIC** individual> <**CLASSIC**-descr>) Adds the information specified in the description to the existing individual, which is then classified and returned.

(**cl-ind-add-concepts** <derived **CLASSIC** individual> <list of told concepts>) Adds the told concepts to the list of concepts which the individual belongs to, and classifies and returns the individual.

(**cl-ind-add-fillers** <derived **CLASSIC** individual> <role> <list of told fillers>) Adds the fillers (told individuals or **HOST** individuals) to the role on the individual, and classifies and returns the individual.

(**cl-ind-close-role** <derived **CLASSIC** individual> <role>) Asserts that the role is closed on the individual. The individual is then classified and returned.

D.7.4 Retraction

(**cl-ind-remove** <derived **CLASSIC** individual> <**CLASSIC**-descr>) Removes everything in the description from the individual, and classifies and returns the individual. Prints warnings for everything from the description which is not in the individual.

(**cl-ind-remove-concepts** <derived **CLASSIC** individual> <list of told concepts>) Removes the told concepts from the list of concepts which the individual belongs to, and classifies and returns the individual.

(**cl-ind-remove-fillers** <derived **CLASSIC** individual> <role> <list of told fillers>) Removes the told fillers, which are either told individuals or **HOST** individuals, from the role on the derived individual, and classifies and returns the individual.

(**cl-ind-remove-all-fillers** <derived **CLASSIC** individual> <role>) Removes all the told fillers from the role on the individual, and classifies and returns the individual.

(**cl-ind-unclose-role** <derived **CLASSIC** individual> <role>) Remove the told fact that the role is closed on the individual.

D.7.5 Batch Processing

The following functions and variables are used for batch processing, which allows CLASSIC to efficiently perform a number of individual retractions all at once. For a description of batch processing, see Section 5.3.

D.7.5.1 Variables

cl-system-status A symbol which is the mode the system is currently in: either normal non-batch processing, or a stage of batch processing. It is used to prevent calling functions while in the wrong mode (i.e., calling **cl-ind-add** while in **:batch-retraction** mode). The possible values are:

:normal non-batch mode;

:batch-retraction during a batch retraction, but before the end;

:batch-end during the processing of a **cl-end-batch-retraction** (this mode should never be seen by the user).

D.7.5.2 Functions

(**cl-start-batch-retraction**) Starts batch retraction. It can only be called if ***cl-system-status*** is **:normal**. It resets ***cl-system-status*** to **:batch-retraction**. All individual retraction calls up until a call to **cl-end-batch-retraction** are recorded but not processed.

(**cl-end-batch-retraction**) Ends batch retraction. It can only be called if ***cl-system-status*** is **:batch-retraction**. It causes all retractions and inferences to be performed for any retractions done since **cl-start-batch-retraction** was called, and then it resets ***cl-system-status*** to **:normal**.

D.7.6 Name Modification

(**cl-create-ind-synonym** <symbol> <classified told individual>) Takes a symbol (a new name) and an existing told individual. Gives the told individual a synonym (the new name). The told individual will still print out as @ti{<old name>}, but @ti{<new name>} can be used to retrieve the individual.

(**cl-ind-change-canonical-name** <symbol> <classified told individual>) Takes a symbol (a name) and an existing told individual. Changes the canonical name of the told individual to the given name, and leaves all existing names. The name may or may not be an existing name for the individual. The told individual will now print out as @ti{<new canonical name>}, but @ti{<old canonical name>} can still be used to retrieve the individual.

D.8 Roles

D.8.1 Predicates

(**cl-role?** <object>) Returns `T` if the object is a role; otherwise `NIL`.

(**cl-attribute?** <role>) Returns `T` if the role is an attribute; otherwise `NIL`. Signals an error if the object is not a role.

D.8.2 Retrieval

(**cl-roles**) Returns a list of all the roles in the knowledge base.

(**cl-named-role** <symbol>) Returns the role with the given name, or `NIL`. Equivalent to @r{`symbol`} (the @r notation is described in Appendix B).

(**cl-role-name** <role>) Returns the canonical name of the role.

(**cl-role-names** <role>) Returns a list of all the names of the role.

(**cl-role-inverse** <role>) Returns the role which is the inverse of the given role, or `NIL`.

(**cl-role-parent** <role>) Returns the role which is the parent of the given role, or `NIL`.

(**cl-role-children** <role>) Returns a list of the roles which are the children of the given role, or `NIL`.

D.8.3 Addition

(**cl-define-primitive-role** <symbol> &key parent inverse (attribute `NIL`) (inverse-attribute `NIL`) inverse-parent) Creates a new role with the given name, and returns the role. The parent keyword specifies the name of the parent role, which must already exist, and which must not be an attribute. The inverse keyword specifies the name of the inverse role, which must not yet exist. The defaults are that the role has no parent or inverse. The attribute and inverse-attribute keywords, which both default to `NIL`, specify whether or not the role and its inverse are attributes, with implicit **AT-MOST** restrictions of `1` (see Section 2.5). The inverse-parent keyword specifies the name of the inverse-role's parent, which must already exist, and which must not be an attribute. An attribute cannot have a parent. Thus, if the attribute keyword is non-`NIL`, the parent keyword must be `NIL`, and if the inverse-attribute keyword is non-`NIL`, the inverse-parent keyword must be `NIL`.

(**cl-define-primitive-roles** <list of symbols> &key parent (attribute `NIL`)) For each symbol, if it is not already the name of a role, creates a role with that name. The parent keyword specifies the name of the parent role for all the roles, if there is one. It must already exist, and it must not be an attribute. If the attribute keyword is non-`NIL`, all the roles are created as attributes; otherwise they are created as non-attributes. If the attribute keyword is non-`NIL`, the parent keyword must be `NIL`. Returns `T`.

D.8.4 Name Modification

(**cl-define-role-synonym** <symbol> <role>) Takes a symbol (a new name) and an existing role. Gives the role a synonym (the new name). The role will still print out as @r{<old name>}, but @r{<new name>} can be used to retrieve the role.

(**cl-role-change-canonical-name** <symbol> <role>) Takes a symbol (a name) and an existing role. Changes the canonical name of the role to the given name, and leaves all existing names. The name may or may not be an existing name for the role. The role will now print out as @r{<new canonical name>}, but @r{<old canonical name>} can still be used to retrieve the role.

D.9 Rules

D.9.1 Predicates

(**cl-rule?** <object>) Returns `T` if object is a rule; otherwise `NIL`.

D.9.2 Retrieval

(**cl-rules**) Returns a list of all the rules in the knowledge base.

D.9.2.1 Converting between names and rules

(**cl-named-rule** <symbol>) Returns the rule with the given name, or `NIL`.

(**cl-rule-name** <rule>) Returns the name of the rule.

D.9.2.2 Retrieval Functions for Rules

(**cl-rule-antecedent** <rule>) Returns the antecedent of a rule, where the antecedent is a classified, derived concept.

(**cl-rule-filter** <rule>) Takes a rule and returns the filter (a derived concept which is not necessarily classified).

(**cl-rule-told-filter** <rule>) Takes a rule and returns the told filter (a told concept which is not necessarily classified).

(**cl-rule-consequent** <rule>) Takes a simple rule and returns the consequent (a derived concept which is not necessarily classified).

(**cl-rule-told-consequent** <simple rule>) Takes a simple rule and returns the told consequent (a told concept which is not necessarily classified).

(**cl-rule-function** <rule>) Returns the function of a description rule or a filler rule.

(**cl-rule-args** <rule>) Returns the arguments of a description rule or a filler rule.

(**cl-rule-role** <rule>) Returns the role of a filler rule.

D.9.3 Creation/Deletion

(**cl-add-rule** <symbol> <classified derived **CLASSIC** concept> <**CLASSIC**-descr>
&key filter comment)

Takes a name (a symbol), an existing *classified* **CLASSIC** concept for the antecedent (see Section 2.1.8), and a **CLASSIC** description for the consequent, and can optionally be given a filter (a **CLASSIC** description), and a comment (a string). It creates a simple rule with the given information, runs it on all appropriate individuals, and returns it (see Section 2.6).

(**cl-add-description-rule** <symbol> <classified derived **CLASSIC** concept> <fn>
&key args filter comment)

Takes a name (a symbol), an existing classified concept for the antecedent, a Common LISP function, and optionally, a list of arguments for the function, a filter (a **CLASSIC** description), and a comment (a string). It creates a description rule with the given information, runs it on all appropriate individuals, and returns it. The function should take as arguments an individual, followed by as many arguments as are specified in the list of arguments. When the rule is fired on an individual, the function gets called with the appropriate arguments, and it produces a **CLASSIC** description, which is then merged into the individual. The function must be written so that once it fires on an individual, it will always produce the same result, even if new information is added to the individual. A filter should be used to keep the rule from firing until all the necessary information is available.

(**cl-add-filler-rule** <symbol> <classified derived **CLASSIC** concept> <role>
<fn> &key args filter comment)

Takes a name (a symbol), an existing classified concept for the antecedent, a role, a Common LISP function, and optionally, a list of arguments for the function, a filter (a **CLASSIC** description), and a comment (a string). It creates a filler rule with the given information, runs it on all appropriate individuals, and returns it. The function should take as arguments an individual, a role, and as many arguments as are specified in the list of arguments. When the rule is fired on an individual, the function gets called with the appropriate arguments, and returns a list of derived individuals. These individuals then get added as fillers of the specified role on the individual. The function must be written so that once it fires on an individual, it will always produce the same list of individuals, even if new information is added to the individual.

(**cl-remove-rule** <rule>) Removes the rule from the knowledge base. This includes removing it from the local rules of its antecedent.

(**cl-remove-all-rules** <classified derived **CLASSIC** concept>) Removes all rules with the given concept as antecedent, and returns the concept.

D.10 Explanation

For a description of explanation in CLASSIC, and for examples and hints for using these functions, see Section 7.

D.10.1 Normalization Explanation Functions

(**cl-exp** <object> &key (role-path `T`) (aspect `T`) (aspect-filler `T`) (all `NIL`)) Takes a classified derived concept, derived **CLASSIC** individual, or simple rule. (For a simple rule, it explains the information in the rule consequent. This also applies to **cl-complete-exp**, **cl-clear-exp**, **cl-set-up-exp**, **cl-get-exp**, and **cl-print-exp**.) First it clears out any pre-existing explanation structures for the object. Then it calculates the explanation for that object, and finally, it prints out the explanation it calculated. If role-path is a list of roles or a single role, then just the explanation along this role-path will be calculated. If role-path is `T` (the default), **cl-exp** will calculate explanation along all role-paths. If aspect is a valid aspect (**ONE-OF**, **TEST**, **PRIM**, **MIN**, **MAX**, **AT-LEAST**, **AT-MOST**, **ALL**, **FILLS**, or **SAME-AS**), then just the explanation of this aspect will be calculated. If aspect is `T` (the default), **cl-exp** will calculate explanations for every aspect. For the aspects **FILLS**, **PRIM**, **TEST**, **ONE-OF**, and **SAME-AS**, if aspect-filler is `T`, all the objects for the specified aspect are explained; otherwise, aspect-filler specifies which filler, primitive, test, **ONE-OF**, or **SAME-AS** restriction to explain. If the all keyword is `NIL` (the default), then the printing function will not print out explanations for anything where the explanation is unknown. If it is non-`NIL` and a piece of information does not have a known source, then "unknown" will be printed.

(**cl-exp-error** <object>) Takes a derived concept or a derived **CLASSIC** individual. If it is an incoherent concept or an inconsistent individual (an error object), then it does the following: it first clears out any pre-existing explanation structures for the object. Then based on the type of error, it calculates an explanation for the information which led to the error. For example, for the error **inconsistent-bounds-conflict**, it would explain where the conflicting **AT-LEAST** and **AT-MOST** restrictions came from.

(**cl-complete-exp** <object> &key (role-path `T`) (aspect `T`) (aspect-filler `T`) (all `NIL`)) Takes a classified derived concept, derived **CLASSIC** individual, or simple rule. Like **cl-exp**, it clears out any pre-existing explanation structures, calculates explanation, and prints out the explanation it calculated. The difference is that it traces the explanation of a fact all the way back to its source, possibly causing it to calculate explanation on other individuals.

(**cl-clear-exp** <object>) Clears all the explanation structures in a derived concept, derived **CLASSIC** individual, or simple rule.

(**cl-set-up-exp** <object> &key (role-path `T`) (aspect `T`) (aspect-filler `T`)) Sets up the explanation for a derived concept, derived **CLASSIC** individual, or simple rule.

(**cl-get-exp** <object> <role-path> <aspect> &optional (<aspect-filler> `T`)) Takes a classified derived concept, derived **CLASSIC** individual, or simple rule, a role-path, an aspect, and an optional argument (an aspect-filler), and returns a list of explanation objects and a boolean value. It follows the role-path along the object, and returns a list of explanation structures which explain the specified aspect. For a simple rule, it explains the rule consequent. For the aspects **FILLS**, **PRIM**, **TEST**, **ONE-OF**, and **SAME-AS**, the optional aspect-filler (default is `T`) specifies which filler, primitive, test, **ONE-OF**, or **SAME-AS** restriction to return the explanation for. If the aspect-filler is not specified for one of these aspects (it is `T`), then a list of lists will be returned, where for each inner list, the car is the particular filler, primitive, test, **ONE-OF**, or **SAME-AS** restriction explained, and the cdr is a list of explanation objects explaining it. Note that if a subsumption function has been used to set up explanation on an object, then one of the five symbols **filler-list**, **prim-list**, **test-list**, **one-of-list**, or **same-as-list** may be used as the aspect-filler, or may be returned when the aspect-filler is T. See the beginning of Section 7.1.2 for a discussion of these symbols. In all cases, the second return value is `T` if the function correctly followed the role-path, and `NIL` otherwise. Thus, if the role-path exists on the object, but there is no explanation object stored for the particular aspect, then `NIL,T` is returned.

(**cl-print-exp** <object> &key (role-path `T`) (aspect `T`) (aspect-filler `T`) (all `NIL`)) Takes a derived concept, derived **CLASSIC** individual, or simple rule, and prints out the explanation for the specified role-path, aspect, and aspect-filler. The explanation is not calculated here—it must have been previously set up by **cl-exp**, **cl-complete-exp**, or **cl-set-up-exp**, or one of the subsumption explanation functions.

(**cl-exp-type** <explanation-object>) Takes an explanation object (a list of explanation objects is returned by **cl-get-exp**), and returns the inference type of the explanation. These inference types are described in Section 7.4.

(**cl-exp-args** <explanation-object>) Takes an explanation object (a list of explanation objects is returned by **cl-get-exp**), and returns the arguments of the explanation.

D.10.2 Subsumption Explanation Functions

(**cl-exp-subsumes-concept** <derived concept> <derived concept>
&key (reason `T`) (origin `NIL`) (all `NIL`))
Takes two derived concepts: a subsuming (more general) concept and a subsumee (more specific) concept, and explains why the subsuming concept *subsumes* the subsumee concept. If the reason keyword is `T` (the default), it explains how each piece of the subsuming concept is satisfied by a piece of the subsumee concept. If the origin keyword is `T` (the default is `NIL`), it explains where each of these corresponding pieces

in the subsumee concept came from. If the all keyword is `NIL` (the default), it only explains the told information from the subsuming concept, ignoring the non-primitive told parents; otherwise, it explains all the derived information. The function generates an error if the first concept does not subsume the second concept.

(**cl-exp-subsumes-ind** <derived concept> <derived individual>
&key (reason `T`) (origin `NIL`) (all `NIL`))

Takes a derived concept and a derived individual, and explains why the concept *subsumes* the individual. If the reason keyword is `T` (the default), it explains how each piece of the subsuming concept is satisfied by a piece of the individual. If the origin keyword is `T` (the default is `NIL`), it explains where each of these corresponding pieces in the individual came from. If the all keyword is `NIL` (the default), it only explains the told information from the subsuming concept, ignoring the non-primitive told parents; otherwise, it explains all the derived information. The function generates an error if the concept does not subsume the individual.

This is the only explanation function that can take a **HOST** individual. Since a **HOST** individual is just the Common LISP object (string, number, etc.), CLASSIC cannot actually store the explanation information it calculates on the individual. However, it will calculate this information and print it out.

(**cl-exp-not-subsumes-concept** <derived concept> <derived concept>)

Takes two derived concepts, and explains why the first concept *does not subsume* the second concept. The function prints a warning if the first concept does subsume the second concept.

(**cl-exp-not-subsumes-ind** <derived concept> <derived **CLASSIC** individual>)

Takes a derived concept and a derived **CLASSIC** individual, and explains why the concept *does not subsume* the individual. The function prints a warning if the concept does subsume the individual.

(**cl-exp-rule-firing** <derived **CLASSIC** individual> <rule>
&key (reason `T`) (origin `NIL`) (all `NIL`))

Takes a derived **CLASSIC** individual, and a rule that fired on the individual, and explains why the rule antecedent subsumes the individual. If the reason keyword is `T` (the default), it explains how each piece of the rule antecedent is satisfied by a piece of the individual. If the origin keyword is `T` (the default is `NIL`), it explains where each of these corresponding pieces in the individual came from. If the all keyword is `NIL` (the default), it only explains the told information on the antecedent, ignoring the non-primitive told parents; otherwise, it explains the derived information.

(**cl-exp-expr-subsumes** <CLASSIC expression> <CLASSIC object>
&key (reason `T`) (origin `NIL`) (all `NIL`))

Takes a **CLASSIC** expression and a **CLASSIC** concept or individual. The expression must parse into a valid concept. Otherwise, an error is generated. If the second argument is a concept, **cl-exp-expr-subsumes** explains why the concept represented by the expression *subsumes* it. If the second argument is not subsumed, an error is generated. If the second argument is an individual, **cl-exp-expr-subsumes** explains why it is an *instance* of the concept represented by the expression. If it is not an instance, an error is generated. If the reason keyword is T (the default), **cl-exp-expr-subsumes** explains how each piece of the subsuming concept is satisfied by a piece of the subsumee. If the origin keyword is T (the default is NIL), it explains where each of these corresponding pieces in the subsumee came from. If the all keyword is NIL (the default), it only explains the told information from the subsuming concept, ignoring the non-primitive told parents; otherwise, it explains all the derived information.

(**cl-exp-expr-missing** <CLASSIC expression> <CLASSIC object>
&key (reason T) (origin NIL) (all NIL))
Takes a **CLASSIC** expression and a **CLASSIC** concept or individual. The expression must parse into a valid concept description. Otherwise, an error is generated. The function **cl-exp-expr-missing** explains why the expression is not a part of the second object's description. There are two possible reasons that can be given. First, the expression may be *redundant*; that is, it *subsumes* an expression in the second object's description. If the reason keyword is T (the default), **cl-exp-expr-missing** explains how the expression is satisfied by a piece of the subsumee. If the origin keyword is T (the default is NIL), it explains where the piece in the subsumee came from. If the all keyword is NIL (the default), it only explains the told information from the subsuming expression, ignoring the non-primitive told parents; otherwise, it explains all the derived information. The second way the expression may be missing is that it is inconsistent with the second object; that is, if the expression were a part of the second object's description, the object would become inconsistent or incoherent.

D.10.3 Pruning Function and Variables

(**cl-exp-set-pruning-variables** &optional <value>) Takes an optional argument, and sets the pruning variables as follows:

- if the argument is unspecified, sets the pruning variables to their default values;
- if it is T, turns on all pruning (this is very close to the default setting);
- if it is NIL, turns off all pruning (prints verbosely).

cl-exp-print-length (default is 3). Sometimes during explanation printing, the expression for a concept is printed. If this variable is non-NIL (it must be an integer of length at least 1), then it acts like the Common LISP variable ***print-length***, and expressions will only be printed up to this length, and then "..." will be printed.

cl-exp-print-level (default is 3). Sometimes during explanation printing, the expression for a concept is printed. If this variable is non-NIL (it must be an integer of length at least 1), then it acts like the Common LISP variable ***print-level***, and expressions will only be printed up to this level, and then # will be printed.

cl-exp-ignore-at-least (default is NIL). If the aspect is T and this variable is T, don't explain **AT-LEAST** restrictions.

cl-exp-ignore-at-most (default is NIL). If the aspect is T and this variable is T, don't explain **AT-MOST** restrictions.

cl-exp-ignore-propagation-filler (default is `T`). For complete explanation, if this variable is `T`, don't explain the filler of a propagation, just the information propagated.

cl-exp-ignore-top-prims (default is `T`). If this variable is `T`, don't explain `CLASSIC-THING` or `HOST-THING`.

cl-exp-ignore-all-prims (default is `NIL`). If this variable is `T`, don't explain any primitives.

cl-exp-ignore-intervals (default is `T`). For complete explanation, if this variable is `T`, don't explain **interval-length-equals-at-least-implies-fillers** (i.e., if fillers got added because of an interval, don't explain where the **MIN** and **MAX** restrictions came from).

cl-exp-ignore-interval-tests (default is `T`). If a concept contains an interval (**MIN** and/or **MAX** restriction), don't explain the tests **atom**, **numberp**, **rationalp**, or **integerp**.

cl-exp-ignore-redundant-propagations (default is `T`). Only explain the most specific propagations. For an example, see Section 7.2.

cl-exp-ignore-redundant-one-ofs (default is `T`). Only explain the final **ONE-OF** restriction, and possibly the most specific **ONE-OF** restrictions, whose intersection generated the final **ONE-OF** restriction.

cl-exp-tests-level (default is 0). Governs the amount of information to present about test subsumption during explanation. Its values can be 0, 1 or 2. A value of 0 means that CLASSIC should indicate only that the subsumer's tests subsume the subsumee's tests in some way. A value of 1 means to show explicitly one subsumed test for each test in the subsuming concept. A value of 2 means to show explicitly every subsumed test for each test in the subsuming concept. For more details, see Section 9.6.1.

cl-complete-exp-ignore-rules (default is `NIL`). In complete explanation on an individual, when a piece of information is explained by a rule firing, the next step is to explain why the rule fired, i.e., why the antecedent of the rule was satisfied. If this pruning variable is `T`, the rule firing will not be explained.

D.11 Enhancement Functions

The following function does not add any new functionality to CLASSIC, but it allows the user to do more complicated things in one step.

(**cl-populate** <**CLASSIC**-descr> <list of symbols>) Takes a concept description (which may just be the name of a concept) and a list of names of individuals, and for each individual, either creates it with the given information, or adds this information to the existing individual. If there is an error while adding the information to an individual, prints the error information on the error stream, but continues to process the rest of the individuals. Returns the list of individuals that were successfully processed. For example,

(cl-populate `'(AND PERSON (AT-LEAST 1 child)) '(Bob Fred))`
creates individuals named `Bob` and `Fred` (if they don't exist) and adds the information to each that he is a person with at least 1 child. If bob could not be a person because of some other disjoint primitive that he belonged to, the information would still be added to fred, and (@i{Fred}) would be returned.

D.12 Meta Information and Interesting Aspects

D.12.1 Describing the Interesting Aspects of Objects

In practice, objects in **CLASSIC** often grow rather complex. Any accurate and useful domain model about real objects inevitably will be filled with many concepts, roles, rules, individuals and interconnections. A concept can have many instances, children, ancestors, rules and any of a number of restrictions. Similarly, an individual can have any number of restrictions, role fillers, primitive ancestors and so on. Printing an object or explaining some derived information about that object can take a lot of time and, when is all is said and done, provide far more information that one can sift through in order to find whatever is of interest.

To address this, **CLASSIC** provides several powerful mechanisms for describing what aspects of an object are interesting and under what circumstances those aspects are interesting. When appropriate, this information is used to prune printed output so that only the information that is interesting is displayed.

(**cl-set-up-concept** <symbol> &key (group-names `'(t)`) (object-filter `'(and)`) (role-path `NIL`) (role-aspect `NIL`) (toplevel-aspect `NIL`) (aspect-before-filter `'(and)`) (aspect-function `'(identity)`) (aspect-after-filter `'(and)`)) Takes the name of a **CLASSIC** concept and a set of values for describing what is interesting about that concept and the conditions under which those things are interesting. Individuals that are instances of this concept will only have the information that is described as interesting displayed when printed with **cl-print-ind-interesting**. Similarly, the concept itself will only have the information described as interesting displayed when printed with **cl-print-concept-interesting**. Subconcepts of this concept will also inherit these interesting properties.

There are two types of keyword arguments to **cl-set-up-concept**: those that describe *which aspects* of an object are interesting and those that describe *under which conditions* those aspects are interesting.

When describing what is interesting about a concept and its instances, one may specify either a toplevel-aspect or a role-aspect and a role-path. If toplevel-aspect is specified then it must be one of **ONE-OF**, **TEST**, **PRIM**, **MIN**, **MAX**, **SAME-AS**, **PARENTS**, **DESCENDANTS**, **INSTANCES**, **DESCENDANT-INSTANCES**, **CHILDREN**, **LOCAL-RULES**, **RULES**, or **COMMENT**.

Permissible role-aspects are **ONE-OF**, **TEST**, **PRIM**, **MIN**, **MAX**, **AT-LEAST**, **AT-MOST**, **ALL-NAMED-CONCEPT**, **FILLS**, **SAME-AS** and `T`. Each of these aspects describes some part of a role-restriction on an object. The role-aspect **ALL-NAMED-CONCEPT** can be used to assert that a role restriction is interesting when it is actually a named concept. The role-aspect value of `T` means simply "all role restrictions" and may be used as a shortcut to assert all of the other role-aspects, including **ALL-NAMED-CONCEPT**.

The keyword argument group-names allows one to "tag" an interesting description with arbitrary symbols or names. When printing out interesting things one can specify which descriptions to use by specifying these names.

Those keywords describing under which conditions an aspect is interesting are: object-filter, aspect-before-filter, aspect-after-filter and aspect-function. The object-filter is a **CLASSIC** expression. The value of an object's aspect will only be printed if the object can "pass" this filter (i.e. is subsumed by or is an instance of the **CLASSIC** expression). Once an aspect has been proven interesting, the values of that aspect (e.g. the list of parents or the fillers of a particular role) are considered. First, each value of the interesting aspect must pass the **CLASSIC** expression aspect-before-filter (much like an object must pass the object-filter). Those that do are then given to the aspect-function. An aspect-function is a lisp expression of the form (**function** $< arg_2 > < arg_3 > ... < arg_n >$)[23]. The aspect values are passed, as a list, as the first argument to the **function** with any additional arguments passed second, third and so on. **Function** will return the list or a subset of it. Those values must then, in turn, pass the aspect-after-filter.

Finally, those values that survive are printed out.

Note that the default values for each of the filters and the aspect-function are such that every aspect and aspect-value will pass.

Also note that repeated calls to **cl-set-up-concept** for a particular concept name and a particular aspect are combined: the object-filters are combined as if by an implicit **AND**. This is also true for the aspect-before-filters and the aspect-after-filters. Similarly, the aspect-functions are combined so as to produce their intersections. A concept inherits all the calls to **cl-set-up-concept** for all of its ancestors.

(**cl-set-up-concept-exp** <symbol> &key (group-names `'(t)`) (object-filter `'(and)`) (role-path `NIL`) (role-aspect `NIL`) (toplevel-aspect `NIL`) (aspect-before-filter `'(and)`) (aspect-function `'(identity)`) (aspect-after-filter `'(and)`)) This is the same as **cl-set-up-concept** except that it describes what is interesting to explain when **cl-print-exp-interesting** is called on an object.

(**cl-set-down-concept** <symbol>) This clears all calls to **cl-set-up-concept** made specifically to concept named by <symbol>. It is as if those calls were never made. Note that this means that any descendants of this concept will "uninherit" these calls as well. On the other hand, this does *not* clear any calls to **cl-set-up-concept** inherited by the concept; those calls still exist.

[23] For a complete list of built-in **CLASSIC** functions as well as instructions for building special-purpose aspect functions, see Appendix D.12.4 and Appendix D.12.5, respectively

(**cl-set-down-concept-exp** <symbol>) This clears calls to **cl-set-up-concept-exp** in exactly the same way that **cl-set-down-concept** clears calls to **cl-set-up-concept**.

(**cl-copy-concept-set-up-to-exp** <symbol>) This procedure takes all the information describing what is interesting to print about a concept's children and instances *at the time of the function call* and also makes it interesting to explain.

(**cl-expand-set-up-concept-exp** <symbol> <descr> <list of roles> <list of roles> <list of symbols> &key (check-start-roles `NIL` (aspect-before-filter `'(and)`) (aspect-function `'(identity)`) (aspect-after-filter `'(and)`)) Takes the name of a concept, a description to be used as an object filter, a list of roles to used as the beginning of a role-path, a list of names to be used as the end of a role path and a list of symbols to be used as role aspects. Each starting role is paired with each ending role to form a role-path and each role-path is in turn paired with each role-aspect (i.e. the cross product of the three lists). These values, along with the values for aspect-before-filter, aspect-function, and aspect-after-filter and are used as arguments to **cl-set-up-concept-exp**. The <descr> is also passed to **cl-set-up-concept-exp** as the object-filter. If check-start-roles is specified, it should be a symbol that names a function suitable to be in the expression (TEST-C <symbol> role-name). It will be used to check each role in the list of starting roles (for an example, see **cl-test-role-filled?**). For a more complete description of how the values are used, see **cl-set-up-concept-exp**.

D.12.2 Printing Interesting Aspects of Objects

(**cl-print-ind-interesting** <individual> &key (interesting-groups `'(t)`) (told `NIL`)) This prints an individual as **cl-print-ind** except that it only prints out those facts about the individual deemed interesting as described by calls to **cl-set-up-concept**. The keyword interesting-groups allows control over which of those descriptions are important. Descriptions can be tagged with arbitrary names. The interesting-groups keyword allows you to specify which descriptions to use for determining interestingness by listing their names. For example, one might label some descriptions with the symbol 'debugging and ask for these descriptions when debugging but use any descriptions labeled with the symbol 'demo for talks and presentations.

(**cl-print-concept-interesting** <individual> &key (interesting-groups `'(t)`) (told `NIL`) (all `'unspecified`)) This prints a concept as **cl-print-concept** except that it only prints out those facts about the concept deemed interesting as described by calls to **cl-set-up-concept**. The keyword interesting-groups allows control over which of those descriptions are important.

(**cl-print-exp-interesting** <individual> &key (interesting-groups `'(t)`)) This explains an object **cl-print-exp** except that it only explains those facts deemed interesting as described by calls to **cl-set-up-concept-exp**. This function performs an implicit call to **cl-clear-exp**. The keyword interesting-groups allows control over which of those descriptions are important. Descriptions can be tagged with arbitrary names. The interesting-groups keyword allows you to specify which descriptions to use for determining interestingness by listing their names. For example, one might label some descriptions with the symbol 'debugging and ask for these descriptions when debugging but use any descriptions labeled with the symbol 'demo for talks and presentations.

D.12.3 Printing and Explaining Interesting Descriptions

With a facility for describing things that are interesting comes the need to understand why something you thought was interesting indeed turned out not to be interesting or why something you thought was uninteresting turned out to be interesting. This is useful for debugging the interesting descriptions, of course, but it can also be a useful tool for understanding the domain hierarchy. Below we describe functions used for printing and explaining interesting descriptions.

(**cl-print-all-interesting-descriptions** <object> &key (skip-filtering `NIL`) (annotate `NIL`) (interesting-groups '(`t`))) This function prints the interesting descriptions that apply to object. This is basically a way to recover the calls made to **cl-set-up-concept** and to determine how they have combined to determine the interesting aspects of a particular object.

If skip-filtering is non-`NIL`, all these descriptions are printed. If it is `NIL` those descriptions of aspects that fail the object-filter will not be printed.

If annotate is non-`NIL` then a description of how a particular description was derived for an aspect is also printed.

(**cl-print-all-interesting-descriptions-exp** <object> &key (skip-filtering `NIL`) (annotate `NIL`) (interesting-groups '(`t`))) This function is like **cl-print-all-interesting-descriptions** except that it recovers calls to **cl-set-up-concept-exp**

(**cl-print-interesting-description** <object> &key (annotate `NIL`) (interesting-groups '(`t`)) (role-path `NIL`) (role-aspect `NIL`) (toplevel-aspect `NIL`)) This function is like **cl-print-all-interesting-descriptions** except that it only prints the interesting description for a particular toplevel-aspect or for a particular role-aspect along a given role-path.

(**cl-print-interesting-description-exp** <object> &key (annotate `NIL`) (interesting-groups '(`t`)) (role-path `NIL`) (role-aspect `NIL`) (toplevel-aspect `NIL`)) This function is like **cl-print-all-interesting-descriptions-exp** except that it only prints the interesting description for a particular toplevel-aspect or for a particular role-aspect along a given role-path.

(**cl-print-exp-interesting-description-not-printed** <object> &key (interesting-groups '(`t`)) (stream `t`) (aspect-fillers `NIL`) (role-path `NIL`) (role-aspect `NIL`) (toplevel-aspect `NIL`)) This function explains why a particular object–or list of objects–that acts as a filler of either a toplevel-aspect or as a filler of a role-aspect along a given role-path is not considered interesting enough to print. If aspect-fillers is `NIL` then all possible fillers are explained.

D.12.4 Built-in Aspect Functions

The mechanism used to describe the interesting aspects of **CLASSIC** objects is powerful; however, that power is limited by the usefulness of the functions that are available to the mechanism to be used as aspect-functions and the various filters. Below, we describe the aspect functions that are provided with the **CLASSIC** system.

(**cl-mark-concept-interesting** <symbol>) This "marks" the concept named by <symbol> as interesting. When the aspect-function **cl-remove-uninteresting-concepts** is used for a particular aspect (e.g. the parent aspect), this concept will be printed if it is a value of that aspect.

(**cl-mark-concept-not-interesting** <symbol>) This "marks" the concept named by <symbol> as uninteresting. When the aspect-function **cl-remove-uninteresting-concepts** is used for a particular aspect (e.g. the parent aspect), this concept will not be printed even if it is a value of that aspect.

(**cl-with-new-concepts-interesting** <lisp expression>*) This macro creates an environment where any concepts created within its scope (i.e. while the lisp expressions are being executed) are "marked" interesting. Thus, when the aspect-function **cl-remove-uninteresting-concepts** is used for a particular aspect (e.g. the parent aspect), these concepts will be printed if they are values of that aspect. This macro can be nested with **cl-with-new-concepts-not-interesting** as well as other calls to **cl-with-new-concepts-interesting**.

(**cl-with-new-concepts-not-interesting** <lisp expression>*) This macro creates an environment where any concepts created within its scope (i.e. while the lisp expressions are being executed) are "marked" as not interesting. Thus, when the aspect-function **cl-remove-uninteresting-concepts** is used for a particular aspect (e.g. the parent aspect), these concepts will not be printed even if they are values of that aspect. This macro can be nested with **cl-with-new-concepts-interesting** as well as other calls to **cl-with-new-concepts-not-interesting**.

(**cl-concept-is-interesting?** <concept>) returns `T` if concept has been "marked" interesting (see **cl-mark-concept-interesting**) and `NIL` otherwise.

(**cl-min-number** <list of numbers>) This aspect function, given a list of numbers, returns a singleton list containing the minimum value.

(**cl-max-number** <list of numbers>) This aspect function, given a list of numbers, returns a singleton list containing the minimum value.

(**cl->** <list of numbers> <number> &optional <symbol> This aspect function, given a list of numbers and another number, returns all the numbers in that list greater than that number. If the optional argument is provided it must be one of the symbols 'zero, 'infinity or 'negative-infinity. In this case, an empty list is treated as if it were zero, infinity or negative infinity, respectively and either an empty list or a list containing `NIL` may be returned.

(**cl->=** <list of numbers> <number> &optional <symbol> This aspect function, given a list of numbers and another number, returns all the numbers in that list greater than or equal to that number. If the optional argument is provided it must be one of the symbols 'zero, 'infinity or 'negative-infinity. In this case, an empty list is treated as if it were zero, infinity or negative infinity, respectively and either an empty list or a list containing `NIL` may be returned.

(**cl-<** <list of numbers> <number> &optional <symbol> This aspect function, given a list of numbers and another number, returns all the numbers in that list less than that number. If the optional argument is provided it must be one of the symbols 'zero, 'infinity or 'negative-infinity. In this case, an empty list is treated as if it were zero, infinity or negative infinity, respectively and either an empty list or a list containing `NIL` may be returned.

(**cl-<=** <list of numbers> <number> &optional <symbol> This aspect function, given a list of numbers and another number, returns all the numbers in that list less than or equal to that number. If the optional argument is provided it must be one of the symbols 'zero, 'infinity or 'negative-infinity. In this case, an empty list is treated as if it were zero, infinity or negative infinity, respectively and either an empty list or a list containing `NIL` may be returned.

(**cl-remove-uninteresting-concepts** <list of objects>) This aspect function, given a list of objects, removes any item that is a concept marked as uninteresting via **cl-mark-concept-not-interesting**.

(**cl-most-specific-concepts** <list of concepts>) This aspect function, given a list of concepts, returns only those concepts from the list that have no children in the list.

(**cl-least-specific-concepts** <list of concepts>) This aspect function, given a list of concepts, returns only those concepts from the list that have no parents in the list.

(**cl-more-specific-than** <list of objects> <symbol>) This aspect function, given a list of objects and the name of a concept, returns only those concepts from the list that are its descendants and only those individuals from the list that are instances of one of its descendants.

(**cl-named-concepts** <list of objects>) This aspect function, given a list of objects, returns only those that are named concepts. Most often used with the `all` aspect.

(**cl-member-of-concepts** <list of objects> &rest <symbols>) This aspect function, given a list of objects and names of concepts, returns only those concepts from the list that are one of the named concepts and only those individuals from the list that are instances of one of the named concepts.

(**cl-member-of-inds** <list of objects> &rest <symbols>) This aspect function, given a list of objects and names of individuals returns only those objects from the list that are one of the named individuals.

(**cl-name-begins** <list of concepts and individuals> <symbol>) This aspect function, given a list of objects and a symbol returns only those objects from the list whose names begin with a substring that matches the string of characters that makes up <symbol>. The symbol NIL is treated as the "empty" string.

(**cl-name-ends** <list of concepts and individuals> <symbol>) This aspect function, given a list of objects and a symbol returns only those objects from the list whose names end with a substring that matches the string of characters that makes up <symbol>. The symbol NIL is treated as the "empty" string.

(**cl-name-contains** <list of concepts and individuals> <symbol>) This aspect function, given a list of objects and a symbol returns only those objects from the list whose names contain a substring that matches the string of characters that makes up <symbol>. The symbol NIL is treated as the "empty" string.

(**cl-name-contains*** <list of concepts and individuals> &rest <symbols>) This aspect function, given a list of objects and a symbol returns only those objects from the list whose names contain substrings that match the string of characters that make up each <symbol> one after the other. The symbol NIL is treated as the "empty" string. In other words, only those objects whose names contain the regular expression "$* < symbol_1 > * < symbol_2 > * \ldots * < symbol_n > *$", where "*" can match any characters including no characters.

(**cl-intersection** <list of objects> &rest <aspect function expressions>) This aspect function, given a list of objects and some aspect function expressions, applies each of the aspect functions to the list of objects and returns the intersection of their return values. Note: there must be at least one aspect function expression.

(**cl-union** <list of objects> &rest <aspect function expressions>) This aspect function, given a list of objects and some aspect function expressions, applies each of the aspect functions to the list of objects and returns the union of their return values. Note: there must be at least one aspect function expression.

(**cl-complement** <list of objects> <aspect function expression>) This aspect function, given a list of objects and an aspect function expression, applies the aspect function to the list of objects and returns all the objects that the aspect function expression did not.

(**cl-compose** <list of objects> &rest <aspect function expressions>) This aspect function, given a list of objects and some aspect function expressions, returns the composition of the aspect function expressions on the list of objects. In other words, the last expression is applied to the list of objects and the return value is passed to the next-to-last expression and that return value is than passed to the previous expression and so on (i.e. ($< expression_1 >$ ($< expression_2 >$ ... ($< expression_1 >$ <list of objects>)))) Note: there must be at least one aspect function expression.

D.12.5 Syntax Checking

You may find it necessary to construct special-purpose functions to be used as aspect-functions. To do this, it is necessary to declare these functions to **CLASSIC** and to follow certain rules. This declaration helps **CLASSIC** to insure that interesting descriptions are syntactically correct.

In particular, all aspect-functions must take a minimum of one argument: a list of the objects that are the fillers of a particular aspect. All aspect-functions must also return a list. Presumably this will be the original list or a subset; however, this is not required if, for example, you want to return the average of a list of numbers (in this case, this average must be returned as a list containing a single number).

Below is a description of the functions used to declare aspect-functions.

(**cl-declare-aspect-function** <symbol> &rest <argtypes>) Takes a symbol that is a name of a function and a symbol that describes an argument type for each argument of the function (except the first one since this is known to be a list of objects).

An argument type is the name of any function that will determine whether an object is of a particular type (e.g. **'integerp** or **'symbolp**). A few special types are provided. The first is **'cl-form**. This type indicates that the value of the argument must be a valid aspect-function call. It is used for functions like **cl-intersection**. There is also the type function '(**&list** <argtype>). Here, <argtype> is simply another argument type. This means that the value of the argument must be a list whose members are all of the type indicated.

The &rest and &optional flags are supported; however, &key is not. The &rest flag acts like '(**&list** <argtype>) and is written the same way. The &optional flag is written by itself and is expected to be followed by more types. For example, one might declare **cl-<** like:

```
(cl-declare-aspect-function cl-< <list of numbers>
                                  <number>
                                  &optional <symbol>)
```

If a function has already been defined, use **cl-declare-aspect-function** to declare it as an aspect-function; otherwise, use **cl-defun-aspect-function**. **Cl-declare-aspect-function** is reserved mainly for built-in functions like **identity**:

(cl-declare-aspect-function 'identity).

(**cl-defun-aspect-function** <function-name> (<required-argument> &rest <argument-argtype pairs>) &body <lisp-body>) This is simply **defun** with typing information included (except for the first argument; this is known to be a list of objects). Instead of providing just an argument name for each argument (as in **defun**), the user must provide an argument type-argument name pair for each argument. These argument types are the same as those for **cl-declare-aspect-function**. For example, **cl-member-of-concepts** is defined:

```
(cl-defun-aspect-function cl-member-of-concepts objects
                                    &rest (symbolp concept-names))
    (let...
    ))
```

Similarly, **cl-union** is defined:

```
(cl-defun-aspect-function cl-union objects
                                   (cl-form form1)
                                   &rest (cl-form forms))
    (let...                        ))
```

Notice that **cl-defun-aspect-function** is designed to look as much like **defun** as possible. Single quotes "”' are not needed for the arguments.

D.13 Extending CLASSIC Using Tests

Test restrictions allow CLASSIC to use procedures to specify concepts (for a complete description of test restrictions, see Section 2.1.4.1). In this way, it is possible to extend the CLASSIC operators available. Unfortunately, procedures are black boxes. Generally speaking, it is not possible for CLASSIC to reason about them as if they were full-fledged operators. This appendix describes several functions that allow CLASSIC to reason more deeply about test restrictions. We strongly recommend reading Section 9 before attempting to use any of these functions.

D.13.1 Test Normalization Functions

Normalization for tests is the process of taking an expression in one form and turning it into another form that is easier to represent and work with. In this section we list functions that allow one to describe for CLASSIC how to normalize test expressions (see also Section 9.1).

(**cl-define-normalize-rule-by-function** <symbol> <function>) Takes a symbol and a function, registering the function as one that normalizes test expressions using the test function <symbol>. By convention, the normalization function for a test function named <symbol> should be named `<symbol>-normalize`.

(**cl-test-normalize** <test expression>) Takes a test expression, finds the registered normalization function, calls it on the test expression, and returns the normalized test expression. If no normalization function exists, the original test expression is returned.

D.13.2 Test Subsumption Functions

In this section we list functions that allow one to describe for CLASSIC the subsumption relationships between test expressions (see also Section 9.2).

(**cl-define-subsumption-rule-by-function** <symbol> <symbol> <function>) Takes two symbols and a function, registering the function as one that determines whether a test expression using the test function named by the first symbol subsumes another test expression using the test function named by the second symbol. By convention, such subsumption functions should be named `<symbol`$_1$`>-subsumes-<symbol`$_2$`>`. If the two symbols are the same, the convention allows the function to be named `<symbol`$_1$`>-subsumes`.

(**cl-test-subsume?** <test expression> <test expression>) Takes two test expressions, finds their registered subsumption function, if one exists, and calls it on the test expressions. If no subsumption function exists, then **cl-test-subsume?** checks for textual equality.

(**cl-test-lists-subsume?** <list of test expressions> <list of test expressions>) Takes two lists of test expressions, and for each expression in the first list, calls **cl-test-subsume?** on it and each expression in the second list. The function returns a list whose first element is a list of explanations for all the calls that returned non-`NIL` and whose second element is a list of explanations for all the calls that returned `NIL`.

(**cl-exp-tests-subsume?** <derived concept> <CLASSIC object> &optional (verbose? `NIL`)) Takes a concept and a concept or individual. The tests of the first argument (the subsumer) are tested to see if they subsume the tests of the second argument (the subsumee). The function **cl-exp-tests-subsume?** returns a list of explanation inferences indicating that each of the subsumer's tests does indeed subsume some test in the subsumee. If there is not a match for every one of the subsumer's tests, `NIL` is returned. If the optional argument `verbose?` is `NIL`, **cl-exp-tests-subsume?** will find only one corresponding test in the subsumee. If `verbose?` is non-`NIL` then **cl-exp-tests-subsume?** will find every corresponding test in the subsumee.

D.13.3 Test Conjunction Functions

In this section we list functions that allow one to describe for CLASSIC how test expressions interact when conjoined (see also Section 9.3).

(**cl-define-conjoin-rule-by-function** <symbol> <symbol> <function>) Takes two symbols and a function, registering the function as one that conjoins a test expression using the test function named by the first symbol and another test expression using the test function named by the second symbol. By convention, such conjunction functions should be named `conjoin-<symbol`$_1$`>-<symbol`$_2$`>`. If the two symbols are the same, the convention allows the function to be named `conjoin-<symbol`$_1$`>s`.

(**cl-test-conjoin** <test expression> <test expression>) Takes two test expressions, finds their registered conjunction function, if one exists, and calls it on the test expressions. If no conjunction function exists, then `T` is returned.

(**cl-test-sets-producing** <test expression> <list of test expressions> &key (skip-singles NIL) (skip-supersets NIL)) Takes a test expression and a list of test expressions. All combinations of the list of test expressions are conjoined in order to see which produce the test expression in the first argument. The test expression may be `NIL`. In this case, **cl-test-sets-producing** looks for incompatible tests. The result of this function is a list of lists. Each sublist is a list of tests that produce the desired result. The keywords `skip-singles` and `skip-supersets` prune the search space. If `skip-singles` is `T`, **cl-test-sets-producing** will not check to see if any of the test expressions in the list of test expressions is exactly the same as the desired result. If `skip-supersets` is `T` then only minimal combinations will be considered. That is, once a group of tests is found that produces the desired result, no superset of that group will be considered. It is wise to set `skip-supersets` to `T` whenever possible because it can dramatically speed up the searching process.

D.13.4 Test Inconsistency Functions

In this section we list functions that allow one to describe for CLASSIC when to use test functions for inconsistency checking (see also Section 9.4).

(**cl-set-test-for-incompatible-test** <symbol>) Takes a symbol naming a test function, and informs CLASSIC that it can be used for incompatibility testing.

E Error Codes

This appendix describes all the error codes that may be returned when running CLASSIC. The error codes are presented alphabetically, although there are really two different types of error codes. The first kind contains the word "conflict" in its name, and represents a "real" error, which causes an individual or concept to contain inconsistent information. The second kind contains the word "error" in its name, and represents a "syntax" error, where the user enters illegal information for a particular function (e.g., illegal names for individuals or concepts, inappropriate types of objects–supplying a **HOST** individual where a **CLASSIC** individual is expected, etc.), or a poorly-defined user function blows up. For each error code, a list of all the arguments passed back (if any) is given. See Section 6 for more information on handling errors in CLASSIC.

The arguments to each error code are listed after the error code name, enclosed within [ ]'s. [ ] means no arguments.

addition-during-batch-retraction-error [ ] Cannot add information to an individual during a batch retraction.

already-defined-conjoin-rule-error [(1) test function name; (2) test function name] CLASSIC has already been given a conjunction rule for determining how to combine a test expression using the first test function with a test expression using the second test function. See Section 9.

already-defined-normalize-rule-error [(1) test function name] CLASSIC has already been given a rule for determining how to normalize a test expression using this test function. See Section 9.

already-defined-subsumption-rule-error [(1) test function name; (2) test function name] CLASSIC has already been given a subsumption rule for determining when a test expression using the first test function subsumesa test expression using the second test function. See Section 9.

ambiguous-object-error [(1) name] There was more than one CLASSIC object with the given name (i.e., a concept and a role might have the same name), and a unique name was expected (for @name, or (**cl-named-object** `<name>`)).

arg-list-error [ ] This occurs when one CLASSIC function calls another one, and the inner function causes an error. The error information from the inner function gets lost. To avoid this error, for each CLASSIC function call which might produce an error, surround it with tests for the return values (see Section 6).

bad-path-error [(1) **SAME-AS** restriction] The paths in a **SAME-AS** restriction must be non-empty lists.

bad-same-as-error [(1) **SAME-AS** restriction] A **SAME-AS** must consist of a list of exactly 2 attribute paths.

changing-name-of-built-in-concept-error [(1) built-in told concept] Attempt to change the name of a built-in concept, e.g., `CLASSIC-THING`.

close-during-batch-error [(1) the value of ***cl-system-status***] Cannot close a role on an individual while in batch mode.

concept-already-exists-error [(1) name] A concept with the given name already exists.

concept-inconsistent-rules-conflict [(1) antecedent concept name] A concept has inconsistent rule consequents.

derived-access-during-batch-error [(1) the value of ***cl-system-status***] Cannot call any function which depends on derived individual information, while in batch mode.

disjoint-prims-conflict [(1) disjoint primitive told concept; (2) disjoint primitive told concept] Attempt to combine two disjoint primitive concepts.

empty-one-of-conflict [ ] A **ONE-OF** restriction is the empty set—no individuals can satisfy this restriction. This may have been caused by no individuals being enumerated in a **ONE-OF** restriction, or by the intersection of two **ONE-OF** restrictions yielding the empty set, or because a **ONE-OF** restriction was filtered by tests or an interval. This error is signaled only if an error is caused at the top level, i.e., (**cl-define-concept** 'CON '(**ALL** age (**ONE-OF**))) does not signal an error since it describes all individuals with no ages.

error-function-blew-up-error [(1) The error hook function; (2) the error code; (3) the arguments to the error-code (may be `NIL`); (4) the incoherent derived object (optional)] One of the error hook functions blew up.

error-hook-error [(1) The non-function] Attempt to add a non-function to the list of error hook functions.

fillers-internal-realm-conflict [(1) the filler individuals] The fillers contain individuals in both realms.

function-blew-up-during-compilation-error [(1) function] A function in a test restriction or rule blew up while being compiled.

hook-function-blew-up-error [(1) hook variable; (2) hook function that blew up; (3) arguments to hook function] A hook function blew up.

hook-not-a-function-error [(1) hook variable; (2) non-function] One of the objects in the list that a hook variable is bound to is not a function.

hook-not-a-list-error [(1) hook variable; (2) non-list] The hook variable is bound to something that is not a list.

illegal-aspect-error [(1) illegal aspect] The aspect must be one of fills, at-least, at-most, all, one-of, test, prim, min, max, or same-as; or T for all aspects (except for **cl-get-exp**, which cannot take `T` for the aspect).

illegal-aspect-filler-error [(1) aspect-filler; (2) aspect] Must have an appropriate aspect-filler based on the aspect. For an aspect of **FILLS**, the aspect-filler must be a derived **CLASSIC** individual or a **HOST** individual. For **PRIM**, the aspect-filler must be a primitive derived concept. For **TEST**, the aspect-filler must be a list containing a function and optional arguments. For **SAME-AS**, the aspect-filler must be a list of 2 attribute paths.

illegal-aspect-for-aspect-filler-error [(1) illegal aspect; (2) aspect-filler] An illegal aspect was specified with an aspect-filler. Can only specify an aspect-filler when aspect is **FILLS**, **PRIM**, **TEST**, or **SAME-AS**.

illegal-classic-operator-error [(1) operator; (2) description] The operator specified in the CLASSIC description is not a legal operator.

illegal-debug-level-error [(1) level] The debug level must be a non-negative integer (for **cl-set-classic-debug-level**).

illegal-host-ind-error [(1) individual] An illegal **HOST** individual given—it must be quoted.

illegal-interval-restriction-error [(1) interval restriction] An interval restriction must be a number.

incompatible-test-conflict [(1) a test expression; (2) an optional programmer-supplied explanation] Within a particular concept description, the test expression can never be true of any object as it asserts some fact that is incompatible with some other assertion within the concept description.

inconsistent-bounds-conflict [(1) role; (2) **AT-LEAST** restriction; (3) **AT-MOST** restriction] A role has inconsistent number restrictions: the **AT-LEAST** restriction is greater than the **AT-MOST** restriction.

inconsistent-interval-conflict [(1) **MIN** restriction; (2) **MAX** restriction] A **HOST** concept has an inconsistent interval: the **MIN** restriction is greater than the **MAX** restriction.

inconsistent-tests-conflict [(1) a list of two test expressions; (2) a stack trace of successful test combinations up until the conflict; (3) an optional programmer-supplied explanation] The two test expressions cannot both be true of an object. Because the two test expressions may be the result of the conjunctions of other tests, the second argument provides a trace of which tests have been successfully conjoined. The trace takes the form `(TESTS-SIMPLIFY (AND` *test1 test2*`)` *test3*`)`. The first two tests are the tests that conjoined and the third is the result of conjoining them. This information is not usually printed for the user during explanation; however, it is returned whenever test conjunction takes place and there is an error. Please see Section 9 for more information. If the programmer provides an explanation, there will be a third argument containing this explanation as a list.

incorrect-number-of-args-error [(1) CLASSIC description] A CLASSIC description has an incorrect number of arguments.

ind-already-exists-error [(1) name] An individual with the given name already exists.

ind-filler-from-closed-role-error [(1) individual; (2) role; (3) told filler (**HOST** individual or told individual), if problem was caused by removing a filler directly with **cl-ind-remove-fillers**; otherwise `NIL`] For an individual, trying to remove a filler from a closed role.

ind-interval-conflict [(1) individual; (2) min; (3) max] An individual is not within an interval.

ind-one-of-conflict [(1) individual; (2) list of individuals in **ONE-OF** restriction] An attempt is being made to combine an individual with a **ONE-OF** restriction, of which the individual is not a member.

ind-test-conflict [(1) individual name; (2) test function; (3) arguments to test function, if any] An individual is inconsistent with a test restriction.

interval-on-classic-ind-error [(1) CLASSIC description (the **MIN** or **MAX** description)] An attempt is being made to put an interval restriction (**MIN** or **MAX**) on a **CLASSIC** individual.

invalid-mode-for-batch-call-error [(1) the value of ***cl-system-status***] The current batch function is being called in an invalid mode.

invalid-test-form-error [(1) a test expression; (2) an optional programmer-supplied explanation] The test expression is syntactically incorrect.

no-inverse-specified-error [(1) role name] Attempt to define a role with an inverse parent or inverse attribute, without specifying an inverse.

non-classic-rule-error [(1) antecedent concept (or its name for **cl-define-concept**)] Attempt to create rules with a non-**CLASSIC** antecedent concept (i.e., a **HOST** concept or `THING`).

non-existent-aspect-filler-error [(1) aspect-filler; (2) object; (3) role-path; (4) aspect] An aspect-filler does not exist on an object, with a given role-path and aspect.

non-existent-role-path-error [(1) role-path; (2) object] A role-path does not exist on an object.

not-a-classic-concept-error [(1) object] An object is not a derived **CLASSIC** concept.

not-a-classic-ind-error [(1) object] An object is not a derived **CLASSIC** individual.

not-a-classic-type-error [(1) illegal type] An illegal type was given where a CLASSIC type was expected (derived-concept, told-concept, derived-individual, told-individual, or role).

not-a-classified-concept-error [(1) object] An object is not a classified derived concept.

not-a-classified-told-concept-error [(1) object] An object is not a classified told concept.

not-a-concept-name-error [(1) object] A symbol is not the name of a concept.

not-a-derived-concept-error [(1) object] An object is not a derived concept.

not-a-derived-ind-error [(1) object] An object is not a derived **CLASSIC** or **HOST** individual.

not-a-derived-object-error [(1) object] An object is not a derived concept or derived **CLASSIC** individual.

not-a-derived-object-or-simple-rule-error [(1) object] An object is not a derived concept or derived **CLASSIC** individual, or a simple rule. This is signaled from the explanation functions.

not-a-function-error [(1) name or object] A function, or in some cases a symbol naming a function, was expected but not given. For example **cl-define-normalize-rule** expects two arguments: a test function name and a function object. If the first argument is not a symbol naming a function, this error will result. If the second argument is not a function, this error will also result.

not-a-list-error [(1) object] An object is not a list.

not-a-number-error [(1) object] An object is not a number (for function **cl-print-updates**, which is no longer supported).

not-a-role-error [(1) object] An object is not a role, or the name of a role.

not-a-rule-error [(1) object] An object is not a rule.

not-a-string-error [(1) object] An object is not a string.

not-a-symbol-error [(1) object] Object is not a symbol.

not-a-told-classic-ind-error [(1) object] An object is not a told **CLASSIC** individual.

not-a-told-concept-error [(1) object] An object is not a told concept.

not-a-told-ind-error [(1) object] An object is not a told **CLASSIC** or **HOST** individual.

not-a-told-object-error [(1) object] An object is not a told concept or told **CLASSIC** individual.

not-an-attribute-error [(1) object] An object is not an attribute.

null-expression-error [ ] Cannot create an object or an ALL restriction with a null expression.

null-name-error [ ] Cannot create an object with a null name.

one-of-internal-realm-conflict [(1) **ONE-OF** restriction description] The **ONE-OF** restriction contains individuals in both realms.

prim-concept-under-thing-error [(1) concept description] Attempt to define a primitive or disjoint primitive concept directly under `THING`.

realm-conflict [ ] There is a realm conflict within the object currently being processed. (Note: this error is signaled if a primitive concept is being created in the **HOST** realm.)

retraction-during-batch-addition-error [ ] Cannot retract information from an individual during a batch addition.

role-already-exists-error [(1) name] A role with the given name already exists.

role-restr-aspect-with-null-role-path-error [(1) role restriction aspect] A role restriction aspect cannot be specified with a null role path.

role-restr-illegal-bound-error [(1) role restriction (i.e., for (**AT-LEAST** `-1 age`), would return `-1`)] A role restriction must be a non-negative integer.

rule-already-exists-error [(1) name] A rule with the given name already exists.

rule-function-blew-up-error [(1) rule function; (2) arguments the function was called on (includes the individual for a descriptor rule, or the individual and role for a filler rule; followed by the optional arguments)] The function from a descriptor or filler rule blew up when run on an individual, because it did something illegal.

specializing-attribute-error [(1) name of new role; (2) parent role (attribute)] It is illegal to create a role that has an attribute as its parent.

specializing-role-as-attribute-error [(1) name of new attribute; (2) name of parent role] It is illegal to create an attribute with a parent role.

startup-function-blew-up-error [(1) The startup hook function] One of the startup hook functions blew up.

startup-hook-error [(1) The non-function] Attempt to add a non-function to the list of startup hook functions.

subsumption-doesnt-hold-error [(1) The non-subsuming concept; (2) the non-subsumed concept or individual] Subsumption doesn't hold between a concept and either a concept or an individual. (This is signaled for the subsumption explanation functions.)

subsumption-holds-error [(1) The subsuming concept; (2) the subsumed concept or individual] Subsumption holds between a concept and either a concept or an individual. (This is signaled for the "why not" subsumption explanation functions.)

syntax-error [(1) CLASSIC description] Illegal syntax has been given for a concept or **CLASSIC** individual definition—a CLASSIC description is neither a symbol nor a list.

test-conjoin-function-blew-up-error [(1) conjunction function that blew up; (2) one of the test expressions to be conjoined; (3) the other test expression to be conjoined] One of the test conjunction functions blew up.

test-function-blew-up-error [(1) test function; (2) individual name; (3) arguments to test function] A function in a test restriction blew up when run on an individual, because it did something illegal. This should not occur if the test functions are all written correctly and attached to concepts that are placed in the correct position in the concept hierarchy (i.e., a function which calls **evenp**, a function that expects an INTEGER, should be attached to a concept which is positioned below the built-in concept INTEGER in the concept hierarchy, so **evenp** will never be called on a non-INTEGER).

test-non-function-error [(1) test "function"] The **TEST-C** or **TEST-H** operator has been given an argument which is not a function.

test-normalize-function-blew-up-error [(1) normalization function that blew up; (2) test expression to be normalized] One of the test normalization functions blew up.

test-subsumption-function-blew-up-error [(1) subsumption function that blew up; (2) proposed subsuming test expression; (3) proposed subsumed test expression] One of the test subsumption functions blew up.

F Built-In Concepts

We now list all the built-in concepts in CLASSIC (see Section 2.1.5 for a description of the built-in concepts).
The first three are THING (the top concept in the hierarchy) and its two children: the disjoint primitive concepts CLASSIC-THING and HOST-THING.

Underneath HOST-THING are the following concepts, which represent Common LISP types of the same names:
CHARACTER, NUMBER, RATIONAL, INTEGER, RATIO, FLOAT, COMPLEX, RANDOM-STATE, FUNCTION, SYMBOL, KEYWORD, SEQUENCE, LIST, NULL, CONS, ARRAY, VECTOR, STRING, STRUCTURE, HASH-TABLE, READTABLE, PACKAGE, PATHNAME, STREAM, and ATOM.

There is a built-in concept under CLASSIC-THING, CONCEPT, which is the parent of all meta individuals (see Section 2.4, and functions **cl-concept-meta-ind** and **cl-concept-meta-ind-concept** in Appendix D.6.2.3).

G Unix Style Interface

CLASSIC includes a command line interface styled after UNIX. This interface consists of a set of commands that can be typed directly at the LISP prompt and can be used to display, create, and manipulate CLASSIC objects.

The interface resembles UNIX in two important ways. First, the commands are styled after (although they do not directly mimic) typical UNIX commands. They consist of a single command name followed optionally by switches, keywords, and additional arguments. Second, many of the commands have the same flavor as typical UNIX commands. For example, at any given time there exists a present working concept (or pwc) analogous to the present working directory in UNIX. The pwc is used by interface commands much in the same way the current directory is used by typical UNIX commands. Its initial value is CLASSIC-THING.

To run the command line interface, the user must call the **cl** function with no arguments:
→ (**cl**)

G.1 Command Syntax

Commands are typed directly at the top level LISP prompt and have precedence over the evaluation of global variables with the same name as the commands themselves. As stated above, commands consist of a command name followed optionally by switches, keywords, and additional arguments. All entities except strings on the command line are case insensitive.

A *switch* is a single letter preceded by the switch character "-". Multiple switches may follow a single switch character as long as they appear next to each other with no intervening spaces. They may also be placed anywhere and in any order on the command line. For example, if the command `command` existed, and had the switches `a`, `b`, and `c`, then commands one, two, three, and four would be legal, and commands five and six would be illegal. Note that commands two and three are equivalent.

1. `command`
2. `command -ab`
3. `command -b -a`
4. `command -ca -b`

*5. `command -d`

*6. `command -a b c`

A *keyword* consists of the switch character "-" followed by a keyword identifier, a space, and the argument of that keyword. Keywords may be placed anywhere on the command line, but the argument of the keyword must immediately follow the keyword. If `command` had the keywords `name` and `number`, then commands one and two would be legal and commands three and four would be illegal.

1. `command -a -name fred -b -number 101 -c`
2. `command -a -name joe`

*3. `command -a -name -b fred`

*4. `command -name`

When a command is parsed, all switches and keywords are located and removed from the command line. The remaining entities are considered additional arguments to the command. These arguments can be LISP symbols, strings, or lists. If `command` required at least one additional argument and could accept at most two additional arguments, then the following commands would be legal.

1. `command twist -ab -name joe`
2. `command -a twist -b -name joe`

3. `command -ab -name joe twist`

4. `command -ab -name twist joe`

Commands one, two, and three are equivalent, and command four is different. In commands one, two, and three `joe` is the keyword argument to `name` and `twist` is the required argument. In command four the reverse is true.

G.2 Command Descriptions

The HELP Command

The interface includes an on-line help facility. The `help` command with no arguments gives a list of the available commands; `help` followed by a command gives extended help on that command. A complete list of the available commands are given below. Note that only the capitalized portion of the command name needs to be typed when referring to the command.

```
Commands:
---------
Add -{fir} [-ph] [-ind x] [-role x] [-fill x] [-exp x] [-con x] [concept]
CC <concept ^>
CLEar
CLOse [-u] individual role
COmpile file
CReate -{cir} [-apv] [-name x] [-exp x] [-con x] [-par x] [-inv x] [concept]
Display [-dir] [object]
Explain [-i] [-aspect x] [-path x] [object]
Graph [-ir] [object]
Help [command]
LIst [-cinpr] [concept]
LOad file
Mode [-px] [state]
PRint [-i] [object]
PWc
Remove -{fir} [-ah] [-ind x] [-role x] [-fill x] [-rule x] [concept]
Version
QUIT

Key:
----
[] - optional switches, keywords, or additional arguments
{} Exactly one of these switches must be specified.
<> At most one of these arguments can be specified.
```

Note: remember that a switch consists of the switch character "-" followed by a single letter, and multiple switches can be represented by the switch character followed by multiple letters. Thus, `[-ph]` means that the `p` and/or the `h` switch may optionally be specified.

The ADD Command

The `add` command allows information to be added to existing CLASSIC objects. Its syntax is as follows:

```
Add -{fir} [-ph] [-ind x] [-role x] [-fill x] [-exp x] [-con x] [concept]

Switches:
---------
-f: Add a filler to a role of an individual.
-i: Add an individual expression to an individual.
-r: Add rules to a concept.
-p: Prompt user for optional information not supplied on the command line.
-h: Assume the filler is a host individual instead of a classic individual
    when there is ambiguity.

Keywords:
---------
-ind:  Used to specify the name of an individual.  The argument must be a
       symbol.
-role: Used to specify the name of a role.  The argument must be a symbol.
-fill: Used to specify the filler for a role.
-exp:  Used to specify an individual expression.
-con:  Used to specify the consequents of a rule.  The argument must be
       a list of expressions.
```

If a filler is being added and there is a classic individual by that name, then the filler is assumed to be the classic individual instead of a host individual unless overridden by the `-h` switch. By default `[concept]` is the pwc. If `-exp` is not given for an individual, and `-p` was not specified, then `-exp` defaults to `[concept]`. Rules are added to `[concept]`.

The CC Command

The `cc` command is used to change the present working concept. Its syntax is as follows:

```
CC <concept ^>
```

If called with no arguments, `cc` changes the pwc to `CLASSIC-THING` (its initial value). If it is called with ^, `cc` changes the pwc to its immediate parent concept. If there is more than one parent, the user will be prompted for a choice. If `cc` is called with a concept, then the pwc will be changed to that concept.

The CLEAR Command

The `clear` command clears the knowledge base of all user-defined objects (it calls **cl-startup**). Its syntax is as follows:

```
CLEar
```

The CLOSE Command

The `close` command changes the closed state of a role on an individual. Its syntax is as follows:

```
CLOse [-u] individual role

Switches:
---------
-u: Uncloses the role.
```

By default, `close` adds the told fact that a particular role is closed on an individual.

The COMPILE Command

The `compile` command is used to compile LISP files. Its syntax is as follows:

```
COmpile file
```

Note that `file` must be a LISP string.

The CREATE Command

The `create` command is used to create concepts, individuals, and roles. Its syntax is as follows:

```
CReate -{cir} [-apv] [-name x] [-exp x] [-con x] [-par x] [-inv x] [concept]

Switches:
---------
-c: Adds a concept to the knowledge base.
-i: Adds an individual to the knowledge base.
-r: Adds a role to the knowledge base.
-a: If a role is being added, make it an attribute.
-p: Prompt user for optional information not supplied on the command
    line.
-v: If an inverse role is being added, make it an attribute.

Keywords:
---------
-name: Used to specify the name of a concept, individual, or role.
       The argument must be a symbol.
-exp:  Used to specify a concept or individual expression.
-con:  Used to specify the consequents of a concept.  The argument must
       be a list of expressions.
-par:  Used to specify the name of the parent of the role.  The argument
       must be a symbol.
-inv:  Used to specify the inverse role of the role.  The argument must
       be a symbol.
```

By default [concept] is the pwc. If -exp is not given for an individual, and -p was not specified, then -exp defaults to [concept].

The DISPLAY Command

The display command is used to graphically display an object. Its syntax is as follows:

```
Display [-dir] [object]

Switches:
---------
-d: Display derived information.
-i: If ambiguity, assume object is an individual.
-r: If ambiguity, assume object is a role.
```

By default [object] is the pwc.

The EXPLAIN Command

The explain command is used to print the origin of information on an object. Its syntax is as follows:

```
Explain [-i] [-aspect x] [-path x] [object]

Switches:
---------
-i: If ambiguity, assume object is an individual.

Keywords:
---------
-aspect: The aspect of interest about the object.  Legal values
         are fills, at-least, at-most, all, one-of, prim, and
         test.  If not specified, all aspects are considered.
-path:   A list of roles denoting a path to the given aspect of
         interest, or T for all paths.
```

By default [object] is the pwc.

The GRAPH Command

The graph command is used to graph the hierarchy under a concept or role. Its syntax is as follows:

```
Graph [-ir] [object]

Switches:
---------
-i: Puts individuals into the graph as well.
-r: If ambiguity, assume object is a role.
```

By default [object] is the pwc. If -r is specified with no [object], the complete set of role hierarchies is graphed.

The LIST Command

The list command is used to list concepts and individuals under a given concept. Its syntax is as follows:

```
LIst [-cinpr] [object]

Switches:
---------
-c: Only concepts are printed.
-i: Only individuals are printed.
-n: If [object] is both a concept and an individual, assume it is
    an individual.
-p: Concepts directly above the pwc are printed instead of those below.
-r: Print out concepts recursively.  Concepts and individuals
    that are indirectly associated with the pwc are printed as well.
```

If called with no arguments, list prints out the immediate concepts and individuals directly under the pwc. By default [object] is the pwc. If both an individual and a concept exist with the name of [object], [object] becomes the concept unless overridden by the -n switch. Since individuals don't have descendants, the -p switch must be given when [object] is an individual.

The LOAD Command

The load command is used to load LISP files. Its syntax is as follows:

```
LOad file
```

Note that file must be a string.

The MODE Command

The mode command is used to change certain operating characteristics of the user interface. Its syntax is as follows:

```
Mode [-epx] [state]

Switches:
---------
-e: Expert Mode. When using the Classic-Emacs interface, if
    expert mode is on, then user interface commands echoed
    to the history buffer are kept in command form.  If expert
    mode is off, echoed commands are converted to their LISP
    function call equivalents. (Note: the Classic-Emacs interface
    will be available soon.)
-p: Pretty Print S Expressions.
```

```
-x: X Windows Graphical User Interface. This mode turns on/off
    the Classic Graphical User Interface when available.
```

[State] can be either ON or OFF, and if not specified, the mode is turned on. Initially, the expert mode and the X windows graphical user interface mode are off, and the mode for pretty printing S expressions is on.

The PRINT Command

The print command is used to print detailed information about classic objects. Its syntax is as follows:

```
PRint [-i] [object]

Switches:
---------
-i: If [object] is both a concept and an individual, assume [object] is
    the individual.
```

If called with no arguments, print will output all information about [object]. By default [object] is the pwc. If both an individual and a concept exist with the name of [object], [object] becomes the concept unless overridden by the -i switch.

The PWC Command

The pwc command is used to print out the present working concept. Its syntax is as follows:

```
PWc
```

The REMOVE Command

The remove command is used to remove information from already existing objects. Its syntax is as follows:

```
Remove -{fir} [-ah] [-ind x] [-role x] [-fill x] [-rule x] [concept]

Switches:
---------
-f: Removes a filler from a role of an individual.
-i: Removes an individual from a concept.
-r: Removes rules from a concept.
-a: If rules are being removed, remove them all.
-h: Assume the filler is a host individual instead of a classic
    individual.

Keywords:
---------
-ind:  Used to specify the name of an individual.  The argument must be
       a symbol.
```

```
-role: Used to specify the name of a role.  The argument must be a symbol.
-fill: Used to specify the filler of a role.
-rule: Used to specify the rule to remove from a concept.  The rule
       name can be obtained by using the PRINT command.
```

If a filler is being removed and there is a classic individual by that name, then the filler is assumed to be the classic individual instead of a host individual unless overridden by the `-h` switch. By default `[concept]` is the pwc. Rules and individuals are removed from `[concept]`.

The VERSION Command

The `version` command is used to print out information on the current version of Classic. Its syntax is as follows:

```
Version
```

The QUIT Command

The `quit` command is used to exit the CLASSIC Read-Eval-Print loop. Its syntax is as follows:

```
QUIT
```

H Description-Logic KRSS Module

The Description-Logic Knowledge Representation System Specification (DLKRSS) is a specification for Description-Logic based knowledge representation systems developed by a group of researchers in description logics.[24]

There is now a DLKRSS module for CLASSIC. This module is an implementation of as much of the DLKRSS as is possible in CLASSIC. It includes a full implementation of the core of the specification plus the incoherent-subsumption extension and the number-restriction extension.

All DLKRSS calls are implemented as macros, and are placed in the krss-classic package. The concept and role names built in to the DLKRSS syntax (namely `TOP`, `BOTTOM`, `NUMBER`, `INTEGER`, `STRING`, and `IDENTITY`) are recognized in DLKRSS descriptions no matter what package they occur in.

The package works by translating DLKRSS calls into the closest equivalent CLASSIC call. Whenever this process is inexact a descriptive warning is printed (provided that CLASSIC warning mode is on). The package uses one special variable ***krss-verbose***. If this variable is non-nil, the translated DLKRSS call is printed before it is executed. The package defines :krss and :krss-classic as features.

The normal way of using the DLKRSS package is to use the krss-classic package, perhaps as follows:

[24]Included in the CLASSIC release is a report which gives more details on the specification: Peter F. Patel-Schneider and Bill Swartout, "Description-Logic Knowledge Representation System Specification", 1993.

```
#+krss-classic (progn (use-package :classic) (use-package :krss-classic)
                      (cl-startup)
                      (setq *krss-verbose* NIL)
                      (cl-set-classic-warn-mode T))
```

I Installing and Loading CLASSIC

There are a few steps you must go through to install CLASSIC. Once CLASSIC has been installed, it is easy to load. We describe here the simplest way to compile and load the CLASSIC files. For a more complete specification of the functions **cl-load-files** and **cl-compile-files**, see Appendix D.1.2. If you are installing CLASSIC on a PC, see Appendix I.3.

CLASSIC is now in a package, the CLASSIC package. Most of the external symbols are exported within the load.lisp file. Thus, once you load the load.lisp file, you may use the CLASSIC package from your current package (with the **use-package** function), and then all the external symbols of the CLASSIC package (function names, variables, etc.) will be available for you to use. If you choose not to use the CLASSIC package (i.e., because you wish to avoid name conflicts between the CLASSIC package's external symbols and the symbols in your package), you can still access the external symbols of the CLASSIC package, but you will need to prefix the symbols by **classic:**, i.e., (**classic:cl-load-files**).

I.1 Installing CLASSIC initially and loading it

1. Place the source files in a single directory. If your LISP does not have the ***load-pathname*** variable defined (see Appendix I.4), or you wish to have your source and binary files in different directories, you should modify classic-init.lisp so that the variables ***classic-source-directory*** and ***classic-binary-directory*** are the directories which contain the CLASSIC source (LISP) and binary files, respectively. This file will load the load.lisp file and reset these variables, so you would load this file instead of the load file, in the directions below. (Otherwise, you don't need this file.) Thus, ***classic-source-directory*** might be
   "/home/resnick/classic/2-1/" for a Sun^{tm},
   "F:user:resnick:classic:2-1:" for a Macintosh^{tm}, or
   "h:>resnick>classic>2-1>" for a Symbolics;
   classic-binary-directory might be
   "/home/resnick/classic/2-1/bin/" for a Sun. Note that you must include the final delimeter character, i.e., for a Sun, you must have the final "/" in the directory path. (Hint for Macintosh users: if you are not sure of the whole path name for where the files are stored, just load the load.lisp file, and LISP will tell you.)

2. Start up Common LISP.
   See Appendix I.4 for what to do now if your LISP is not CLTL-2 compatible.

3. Load the load.lisp file.

4. Use the CLASSIC package:
   → (**use-package** :CLASSIC)

(Note: this step is not required, but before omitting it, be sure to read the note at the beginning of this appendix.)

If you have been given all the compiled CLASSIC files (only for Macintosh users),

5. (a) Compile the new load.lisp file, if you changed it.
   (b) Load all the compiled files:
   → (**cl-load-files**)
6. Otherwise, you need to compile all the source files:
   (a) Load all the CLASSIC LISP files as follows:
   → (**cl-load-files** :lisp? t)
   (b) Compile all the CLASSIC files as follows:
   → (**cl-compile-files**)
   (c) Load all the compiled files:
   → (**cl-load-files**)
7. For both of the above, run the command line interface (optional):
   → (**cl**)

Notes on compiling CLASSIC (you may ignore this): like most LISP systems, CLASSIC can be sped up by using more compiler optimizations. For most uses of CLASSIC, this is not necessary. However, if speed is extremely important to your application, you may wish to compile CLASSIC with more aggressive optimization settings of the compiler. For example, you may wish to execute the command

```
(proclaim '(optimize (speed 3) (safety 1) (compilation-speed 0)))
```

before calling **cl-compile-files**. (See the documentation of your particular Common LISP implementation for more information.)

Warning: we have fixed all the known bugs in CLASSIC, but you should be aware that if you do come across a bug in CLASSIC, and the CLASSIC code has been compiled with the above compiler optimization settings, you may experience unusual behavior. In particular, your LISP image may become corrupted in hard to decipher and/or non-recoverable ways.

I.2 Running CLASSIC after it has been installed

1. Start up Common LISP.
2. Load either the load.lisp file or the binary "load" file (load.bin, load.fasl, etc., depending on your machine).
3. Use the CLASSIC package:
   → (**use-package** :CLASSIC)
   (Note: this step is not required, but before omitting it, be sure to read the note at the beginning of this appendix.)

4. Load all the compiled files:
   → (**cl-load-files**)

5. Run the command line interface (optional):
   → (**cl**)

I.3 Installing CLASSIC on the PC

CLASSIC has been modified to run on the PC under Allegro CL/PC 2.0. Note that this lisp will generate a large number of warnings when the uncompiled code is first loaded (the warnings are harmless). To run CLASSIC on the PC, you need to do the following, in addition to the above instructions:

- Rename the extensions on the source files from .lisp to .lsp. There is a rename file, which is included in the src directory. You can run this file on UNIX to rename the source files.

I.4 Compatibility with non-CLTL-2 Common LISPS

If your LISP follows the Common LISP standard (as described in "Common Lisp, The Language, 2nd edition"), it should have the variable ***load-pathname*** defined. If this is the case, then the variables ***classic-source-directory*** and ***classic-binary-directory*** default to the directory where the load.lisp file is located. Thus, if you plan to store the source and binary files in the same directory, you do not need to modify these variables in the load.lisp file.

Within CLASSIC, we make use of the extended **loop** macro that is specified as part of the Common LISP standard. We have included the **loop** macro code in the file ansi-loop.lisp. If you do not have the **loop** macro, you will need to compile this file, and you will need to load it before you load or compile CLASSIC.

I.5 Name Conflicts

We have tried to eliminate name conflicts between CLASSIC's external symbols and system-specific symbols that the LISP implementation puts in the COMMON-LISP-USER package. If such a conflict occurs in your Common LISP implementation, please let us know about it. The conflict will most likely occur during the installation process.

J Release Notes

J.1 Release Notes for CLASSIC 2.3

The major additions/changes between CLASSIC 2.2 and CLASSIC 2.3:

- modified CLASSIC to run on the PC under Allegro CL/PC 2.0. This involved, among other things, renaming all the CLASSIC files so they are at most 8 characters in length. Disclaimer: we have only tested it under Allegro CL/PC 2.0 Common Lisp. This is our first release of classic that works on a PC, and it has not been extensively tested in that environment.

- updated CLASSIC to run on the Mac under MCL 3.0
  - When loading and compiling CLASSIC for the first time, the incremental compiler will generate lots of warnings, which you can ignore. It may take up to 2 hours on slower macs to load and compile.
- added a subject index to the manual, in addition to the function index;
- added the function **cl-exp-error** to explain the information which led to an error;
- added test extensions to allow users to define how two test restrictions are related to each other (for more details, see Section 9):
  - a user can define when one test restriction subsumes another;
  - a user can define how two test restrictions simplify into a single test restriction;
  - added facilties for explanation to work for new test extensions in a user-definable way;

  Note: we have the complete examples of the uses of the test extensions that are referred to in the manual. If you are interested in seeing them, please contact Deborah McGuinness at dlm@research.att.com.
- added a meta language to allow users to state what is interesting about objects (for more details, see Section 8):
  - the functions **cl-print-interesting** and **cl-print-exp-interesting** use the meta language specification to determine what to print or explain, respectively;
  - meta information can now handle much of the work that was done by the pruning variables (see Section 7.2). If a user wants to make sure that the pruning variables do not interact with meta information, she can call (**cl-exp-set-pruning-variables** `NIL`);
  - the default settings for ***cl-exp-ignore-at-least*** and ***cl-exp-ignore-at-most*** have been changed to `NIL` in order to avoid interactions with meta information;
- we now include a demo of CLASSIC which helps the user configure a home entertainment system. To use its full functionality, you must have Allegro Common Lisp 4.2, and CLIM 2.0, but the demo can be run in a limited fashion in a Common Lisp without CLIM. See the Readme-stereo-demo file in the top-level classic directory, for more information;
- Chris Welty (weltyc@cs.vassar.edu) has written an HTML interface to Classic which works, currently, only with MCL (Macintosh Common LISP). For more information, contact http://www.cs.vassar.edu/faculty/welty/classic-dist.html;
- modified the command line interface so that the full name of the command is required, rather than a prefix of the command, in order to reduce potential package conflicts.

Note: We have run CLASSIC in a number of Common Lisp implementations, including: Allegro 4.2, Lucid 4.1, Allegro CL/PC 2.0, MCL 3.0. It has been most thoroughly tested in Allegro 4.2, because that is our development environment. If you want to find up-to-date information about available Lisp implementations, you can look at the lisp FAQ, at http://www.cs.cmu.edu/Web/Groups/AI/html/faqs/lang/lisp/part4/faq.html.

Note: we no longer support compatibility between the current version of CLASSIC and versions prior to 2.0.

J.2 Release Notes for CLASSIC 2.2

The major additions/changes between CLASSIC 2.1 and CLASSIC 2.2:

- added hooks into CLASSIC (see Appendix D.3);
- added a dumping facility for dumping told information (see Appendix D.4);
- added the capability for using dependencies returned by test functions (see the discussion of dependencies for test functions at the end of Section 2.1.4.1);
- enhanced explanation: complete explanation; pruning capabilities; explain why subsumption doesn't hold (see Section 7);
- implemented the KRSS syntax for CLASSIC (see Appendix H);
- added the variable ***classic-print-right-margin***, which sets the right margin for the object printing functions (see Appendix D.2.1).

J.3 Release Notes for CLASSIC 2.1

This section discusses the major additions and changes between CLASSIC 1.0/1.2 and CLASSIC 2.1. It refers to sections of the manual which you can read to get more information.

There is a CLASSIC compatibility package which allows any description from the description language of CLASSIC 1.0 to work in CLASSIC 2.1. It converts uses of the **PRIMITIVE** and **DISJOINT-PRIMITIVE** operators of CLASSIC 1.0 into the appropriate functions in CLASSIC 2.1, and does any necessary modifications needed because of the changes that have been made to primitives and disjoint primitives. For details of the changes, see Appendix J.3.2.1 below. The compatibility package is turned on and off with the function **cl-run-compatibility-package** (see Appendix D.1.2).

J.3.1 Language Additions

J.3.1.1 Complex Roles: Role Hierarchies and Role Inverses Roles are now complex objects. They can be part of a hierarchy, and they can have inverses. The function **cl-define-role** has been replaced by **cl-define-primitive-role**. The function **cl-define-roles** has been replaced by **cl-define-primitive-roles** (see Section 2.5).

J.3.1.2 Rules There are now 3 kinds of rules. The simple rules of CLASSIC 1.2, and description and filler rules. These two new kinds of rules allow you to define a function which will run when a rule is fired on an individual, and will calculate information to merge into the individual (either a CLASSIC description, or a list of fillers for a role, depending on the kind of rule), based on what is currently known about the individual. A rule is now required to have a name. A rule may have a filter, which is a CLASSIC description which determines whether or not the rule will fire on an individual, and it may also have a comment (see Section 2.6).

J.3.1.3 Explanation We now have an explanation facility in CLASSIC which allows you to get an explanation of where a particular fact came from (which inferences caused it to be there). We also explain why a subsumption relationship holds between two objects, and why a rule fired on an individual (see Section 7).

J.3.1.4 Error Handling There are some changes in error handling. When an error is caused, CLASSIC now returns an "error object" which contains all the inconsistent information, and combined with the explanation facility, this allows the user to understand why the error occurred (see Section 6).

J.3.1.5 Batch Processing We now have a batch processing mode for retraction for efficiency, in which CLASSIC delays all processing of information until the batch mode is exited (see Section 5.3).

J.3.1.6 Intervals There is a new **HOST** construct called an interval, which basically requires instances to be within a range of numbers. The **MIN** operator specifies the lower bound, and the **MAX** operator specifies the upper bound. A **HOST** expression can contain a **MIN** restriction, a **MAX** restriction, or both (see Section 2.1.4.2).

J.3.1.7 Error Hooks and Startup Hooks The user may define functions which get called every time an error is generated (error hooks), or every time **cl-startup** is called (startup hooks) (see Appendix D.3.1).

J.3.1.8 Temporary Concepts The user can define temporary concepts, which are not part of the subsumption hierarchy, but which most of the user functions can operate on (see Section 2.1.6).

J.3.1.9 Meta Individuals The user can associate a meta individual with any derived concept, to store information about the extension of the concept (any information currently known about the concept, such as number of instances or average age) (see Section 2.4).

J.3.2 Language Changes

J.3.2.1 Changes in Primitive and Disjoint Primitive concepts There are a number of changes to the CLASSIC language. The **PRIMITIVE** and **DISJOINT-PRIMITIVE**

operators are no longer part of the language. There are now special functions to define primitive and disjoint primitive concepts: **cl-define-primitive-concept** and **cl-define-disjoint-primitive-concept**, respectively. **cl-define-concept** is still used to define regular (defined) concepts.

There is no longer the notion of an index when defining a primitive or disjoint primitive concept (the group-index is still used for disjoint primitive concepts). When one of these concepts is created, it is given a name (as before), and the only way to refer to the identical concept again is by its name.

When a disjoint primitive concept is defined, it is given a list of grouping indices. These are all groups in which it is known to be disjoint from all other members. The group indices are now global, instead of local under a given parent, so you can define a disjoint primitive concept as being disjoint from some other non-related disjoint primitive concept (see Section 2.1.6).

J.3.2.2 Told and Derived Concepts and Individuals We now have the notions of told and derived concepts and individuals, where a told object contains exactly the information the user told CLASSIC about the object, and a derived object contains all the information that CLASSIC derived about the object, using a number of inferences. There are different functions which operate on told and derived objects. Thus, told role restrictions are retrieved from a told concept, while derived role restrictions are retrieved from a derived concept. The user functions have been consolidated so that a number of functions can operate on both told concepts and told individuals, and others operate on both derived concepts and derived individuals. As before, we have the notion of an unclassified concept, which might be a rule consequent or an **ALL** restriction. There are functions that translate between told and derived objects, but these do not work for unclassified concepts (see Section 2.3).

The @ reader macro now also retrieves told concepts and individuals, with @`tc`{name} and @`ti`{name}, respectively. Told concepts and individuals print out as @`tc`{name} and @`ti`{name}, respectively (see Appendix A).

J.3.2.3 Attributes and SAME-AS restrictions In CLASSIC 1.2, attributes had number restrictions of **AT-LEAST** `1` and **AT-MOST** `1`. In CLASSIC 2.1, attributes still have **AT-MOST** restrictions of `1`, but they now have **AT-LEAST** restrictions of `0`. However, when a **SAME-AS** restriction is asserted, all the attributes in both the paths of the **SAME-AS** restriction get **AT-LEAST** restrictions of `1`.

J.3.2.4 Other Changes Some built-in **HOST** concepts have been removed for simplification.

J.3.3 New Implementation

CLASSIC 2.1 has a totally new implementation. We now have a simple recursive-descent parser. The data structures incorporate some sharing of information between different objects. We use a marking algorithm for classification.

J.3.4 Language Description and Functions

For a specification of the language, see Appendix A. For a description of all the CLASSIC functions, see Appendix D.

Index

Index