Living with CLASSIC:

When and How to Use a KL-ONE-Like Language

Ronald J. Brachman
Deborah L. McGuinness
Peter F. Patel-Schneider
Lori Alperin Resnick

AT&T Bell Laboratories
Murray Hill, NJ

Alexander Borgida

Rutgers University
New Brunswick, NJ

Appears in *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, edited by John F. Sowa: Morgan Kaufmann Publishers, San Mateo, CA, 1991, pp. 401–456.

Abstract

CLASSIC is a recently-developed knowledge representation system that follows the paradigm originally set out in the KL-ONE system: it concentrates on the definition of structured concepts, their organization into taxonomies, the creation and manipulation of individual instances of such concepts, and the key inferences of subsumption and classification. Rather than simply presenting a description of CLASSIC, we complement a brief system overview with a discussion of how to live within the confines of a limited object-oriented deductive system. By analyzing the representational strengths and weaknesses of CLASSIC, we consider the circumstances under which it is most appropriate to use (or not use) it. We elaborate a knowledge-engineering methodology for building KL-ONE-style knowledge bases, with emphasis on the modeling choices that arise in the process of describing a domain. We also address some of the key difficult issues encountered by new users, including primitive vs. defined concepts, and differences between roles and concepts, as well as representational "tricks-of-the-trade," which we believe to be generally useful. Much of the discussion should be relevant to many of the current systems based on KL-ONE.

Contents

1 Introduction

Work on the KL-ONE Knowledge Representation System [Brachman and Schmolze, 1985] in the late 1970's inspired the development of a number of frame-based representation systems. These systems have all embraced the ideas of frames as structured descriptions, differentiation between terminological and assertional aspects of knowledge representation, and the central nature of subsumption and classification inferences. At this point there are at least a dozen systems with this shared philosophy and heritage, with widespread international distribution and much ongoing development. All told, there is a large and growing population of users of "KL-ONE-like systems."

While the KL-ONE family has garnered its share of technical publications, virtually all of its literature has described technical details of language design, inference complexity, and semantics. One key issue, of concern to the growing community of users, has remained relatively ignored:[1] how does one go about developing a knowledge base with one of these languages? It is one thing to understand the syntax and semantics of a formal knowledge representation language, but quite another to comprehend how to take a complex domain and represent it appropriately with the constructs afforded by the language.

In this chapter, we attempt to capture some of the lore of building knowledge bases in KL-ONE-like systems. We do this in the context of CLASSIC, a new frame-based description system inspired by KL-ONE and most immediately descended from KANDOR [Patel-Schneider, 1984] (and, as it turns out, closely related to BACK [Peltason *et al.*, 1987]). CLASSIC adopts the point of view that a knowledge base can be treated as a deductive database, in this case one with an object-centered flavor. Because of its intended role as a database-style repository, CLASSIC intentionally limits what the user can say. As a benefit, all inferences can be done in a timely manner. All KL-ONE-like languages are limited in some way, and learning to live with such limitations is one of the keys to making good use of these systems in knowledge-based applications.

CLASSIC has a number of novel features that distinguish it from other KL-ONE-like systems, but here we concentrate less on interesting new developments in the language and focus instead on how to make good use of it.[2] To that end, we first give a brief introduction to the formalities of CLASSIC. We then address the key issue of when a system like CLASSIC is appropriate for an application and when it is not. While we can not give a comprehensive formula for when to use the system, we have tried to give some insight into its strengths and weaknesses, and thus which applications may be best suited to its abilities.

Since CLASSIC and some other KL-ONE-like systems emphasize certain issues relating to terminology and classification that are not common in other KR systems, there tend to be a number of subtle ideas that a user must grasp before he or she can make best use of such systems. Therefore, we address ourselves to several important ideas that may be difficult for the novice user of CLASSIC. These involve, among other things, the differences between primitive and defined concepts and some differences in working with concepts and individuals. We also address the perennial issue of when to make something a concept or a role. Subsequently, we present some guidelines for developing CLASSIC knowledge bases,

[1]An exception is a recent paper on how to build medical knowledge bases in the NIKL language [Senyk *et al.*, 1989]. Discussion regarding "ontological engineering" in CYC [Lenat and Guha, 1990] is also somewhat relevant here.

[2]The interested reader is referred to [Borgida *et al.*, 1989] for details on CLASSIC and its novel contributions.

including a sketch of a knowledge engineering methodology that has worked for us in recent applications. Finally, we offer some "tricks of the trade" for CLASSIC users—some tips on ways to represent certain information that are not obvious from the syntax of the language. For example, judicious use of "test" concepts and CLASSIC rules can provide a facility for integrity checking. All in all, we try to give the potential user an idea not so much of what CLASSIC is, but rather how best to live with it and make it work well in an application.

2 The CLASSIC Knowledge Representation System

CLASSIC[3] is a frame-based knowledge representation system, i.e., its primary means of representation is in describing objects as opposed to asserting arbitrary logical sentences. It allows the user to make assertions about objects (e.g., "Kalin-Cellars-Semillon is a wine," and "Mary drinks Marietta-Old-Vines-Red") and to describe classes of objects (e.g., "a wine made from Cabernet-Sauvignon and Merlot grapes"). The frames in CLASSIC—which we call "concepts"—are interpreted as descriptions rather than assertions. Thus, if we define a wine as a drink with a number of other properties, then being a drink is a necessary part of being a wine, and no wine can violate this requirement.

There are three kinds of formal objects in CLASSIC:

- *concepts*, which are descriptions with potentially complex structure, formed by composing a limited set of description-forming operators (e.g., `WHITE-FULL-BODIED-WINE`[4] might represent the concept of a `WINE` whose `color` property is restricted to being `White` and whose `body` is `Full`); concepts correspond to one-place predicates, and thus are applied to one individual at a time;

- *roles*, which are simple formal terms for properties (e.g., `grape` might represent the grape(s) a wine is made from); roles correspond to two-place predicates, and are used to relate two individuals at a time. Roles that must be filled by exactly one individual are called *attributes* (e.g., `color` might be an attribute representing the color of a wine);

- *individuals*, which are simple formal constructs intended to directly represent objects in the domain of interest; individuals are given properties by asserting that they satisfy concepts (e.g., "`Chardonnay` is a `GRAPE`") and that their roles are filled by other individuals (e.g., "`Kalin-Cellars-Semillon`'s `color` is `White`").

Concepts and individuals in CLASSIC are divided into two realms: *CLASSIC* and *HOST*. *CLASSIC* concepts are used to represent classes of real-world individuals of a domain, while *HOST* concepts are used to describe individuals in the implementation language (currently COMMON LISP), such as numbers and strings. We treat *HOST* concepts and individuals differently from their *CLASSIC* counterparts by not allowing them to have roles (e.g., we cannot attach any properties to the integer 3).

Concepts and individuals are put into a taxonomy, or hierarchy. A more general concept will be above a more specific concept in the taxonomy. For example, if there were a concept for "a wine made from Cabernet-Sauvignon and Merlot grapes," then this would

[3] CLASSIC stands for "**CLASS**ification of **I**ndividuals and **C**oncepts." It has a complete implementation in COMMON LISP.

[4] Throughout this chapter, we use the following orthographic conventions:

`CONCEPT-NAME`: typewriter font, upper case;
`Individual-Name`: typewriter font, capitalized;
`role-name`: typewriter font, lower case;
REALM: slanted, upper case;
function-name: boldface roman, lower case;
CLASSIC-OPERATOR: boldface roman, small capitals.

be a more specific concept than "a wine made from at least one grape," because the first concept describes wines made from at least two grapes. In the taxonomy, individuals are found underneath all the concepts that they satisfy. For example, the individual `Kalin-Cellars-Semillon`, which happens to be a wine whose color is white, would be under the concept `WHITE-WINE` in the taxonomy. To maintain this taxonomy CLASSIC also determines the derivable properties of all individuals and concepts—inheriting properties from more-general descriptions as well as combining properties as appropriate.

There are numerous deductive inferences that CLASSIC provides:

- *completion*: logical consequences of assertions about individuals and descriptions of concepts are computed; there are a number of "completion" inferences CLASSIC can make:
  - *inheritance*: restrictions that apply to instances of a concept must also apply to instances of specializations of that concept; in a sense, then, properties are "inherited" by more specific concepts from those that they specialize;
  - *combination*: restrictions on concepts and individuals can be logically combined together to make narrower restrictions;
  - *propagation*: when an assertion is made about an individual, it may hold logical consequences for some other, related individuals; CLASSIC "propagates" this information forward when an assertion is made;
  - *contradiction detection*: it is possible to accidentally assert two facts about an individual that are logically impossible to conjoin together; CLASSIC detects this kind of contradiction;
  - *incoherent concept detection*: it is possible to accidentally give a concept some restrictions that combine to make a logical impossibility, thereby not allowing any instances of the concept to be possible; CLASSIC detects this kind of inconsistent description;
- *classification and subsumption*:
  - *concept classification*: all concepts more general than a concept and all concepts more specific than a concept are found[5];
  - *individual classification*: all concepts that an individual satisfies are determined;
  - *subsumption*: questions about whether or not one concept is more general than another concept are answered (this is important during concept classification);
- *rule application*: simple forward-chaining rules have concepts as antecedents and consequents; when an individual is determined to satisfy the antecedent of a rule, it is asserted to satisfy the consequent as well.

CLASSIC has a uniform, compositional language, with term-forming operators for creating descriptions of concepts and individuals. The grammar for this language can be found in Figure 1 (we discuss the operators below). Note that individuals can be described with the same expressiveness as concepts. Information can be added to existing individuals, and information can also be retracted from them, with the appropriate consequences.

[5]Note that object-oriented programming languages usually have inheritance, but not classification.

<concept-expr> ::= `THING` | `CLASSIC-THING` | `HOST-THING` |
<concept-name> |
(**AND** <concept-expr>$^{+}$) |
(**ALL** <role-expr><concept-expr>) |
(**AT-LEAST** <positive-integer><role-expr>) |
(**AT-MOST** <non-negative-integer><role-expr>) |
(**FILLS** <role-expr> <individual-name>$^{+}$) |
(**SAME-AS** <attribute-path><attribute-path>) |
(**TEST-C** <fn><arg>*) |
(**TEST-H** <fn><arg>*) |
(**ONE-OF** <individual-name>$^{+}$) |
(**PRIMITIVE** <concept-expr> <index>) |
(**DISJOINT-PRIMITIVE** <concept-expr> <group-index> <index>)
<individual-expr> ::= <concept-expr>
<concept-name> ::= <symbol>
<individual-name> ::= <symbol> | <cl-host-expr>
<role-expr> ::= <mrole-expr> | <attribute-expr>
<mrole-expr> ::= <symbol>
<attribute-path> ::= (<attribute-expr>$^{+}$)
<attribute-expr> ::= <symbol>
<cl-host-expr> ::= <string> | <number> |
'<CommonLISP-expr> | (quote <CommonLISP-expr>)
<fn> ::= a function in the host language (Common LISP) with three-valued logical return type
<arg> ::= an expression passed to a test function
<index> ::= <number> | <symbol>
<group-index> ::= <number> | <symbol>

Figure 1: The CLASSIC Grammar.

We should add that we have taken the approach in CLASSIC that a knowledge representation system should be small and simple (i.e., limited in expressive power), so that its response time is quick, and thorough inference can be peformed. Thus, a user cannot expect to program arbitrary computations in CLASSIC. One should envision CLASSIC as being one component within a larger application system, where it would be used to represent the domain knowledge of the system and calculate a limited set of domain-independent inferences from that knowledge. Other modules in the system would be responsible for the more complicated inferences relating to the particular domain and task.

2.1 Knowledge Base Components

The CLASSIC operators are used to form conjunctions, role restrictions, test restrictions, enumerated concepts, and primitive and disjoint primitive concepts. The typical way of describing a new concept or individual in CLASSIC is to give a list of more general concepts (or in the case of an individual, a list of concepts that are satisfied by the individual), and then a list of restrictions that specify the ways in which this new concept or individual differs from these more general concepts. At the end of this subsection, we also discuss the rule component of CLASSIC.

2.1.1 Named Concepts and Conjunction

The simplest type of concept expression is a single symbol designating a concept. CLASSIC starts off with a number of built-in named concepts, including `THING`, `CLASSIC-THING`, `HOST-THING`, and concepts for each of the COMMON LISP types.[6] These names can be used in other concept expressions to build up complex definitions. While the user can create a new name and make it directly synonymous with an existing one, the simplest useful means of building a compound concept expression is the **AND** operator, which creates a new concept that is the conjunction of the concepts given as arguments. For example, if `WHITE-WINE` and `FULL-BODIED-WINE` are two concepts that have been previously defined, we can define their conjunction as

```
(AND WHITE-WINE FULL-BODIED-WINE)
```

and call it `WHITE-FULL-BODIED-WINE`. This name can then be used in later concept definitions. Note that the **AND** operator can be applied to *any* concept expressions (as long as any names are defined before they are used), not just simple named ones (see Section 2.1.2*ff* for examples).

2.1.2 Role Restrictions

The five operators **ALL**, **AT-LEAST**, **AT-MOST**, **FILLS**, and **SAME-AS** form expressions known as *role restrictions*, and can be used only in *CLASSIC* concepts and individuals, not in their *HOST* counterparts. As specified in the grammar, a role restriction is itself a well-formed concept.

[6] Technically, `THING`, `CLASSIC-THING`, and `HOST-THING` are primitive concepts, with the latter two being disjoint (see Section 2.1.3). The concepts for the COMMON LISP types are formed using the **TEST-H** construct (see also Section 2.1.3), so that all instances of them can be recognized automatically.

A universal value restriction, or **ALL** restriction, specifies that all the fillers of a particular role must be individuals described by a particular concept expression. For example, a `CALIFORNIA-WINE` might be defined as a wine whose region is a California region, where the California regions are Napa Valley, Sonoma Valley, etc. The `region` role restriction would be written

(**ALL** `region CALIFORNIA-REGION`).

AT-LEAST and **AT-MOST** restrictions restrict the minimum and maximum number of fillers allowed for a given role on a concept or individual. For example, part of the definition of a wine might be that it is made from at least one kind of grape, which would be written

(**AT-LEAST** `1 grape`),

where `grape` is a role.

The **FILLS** operator specifies that a role is filled by some specified individuals (although the role may have additional fillers). For example, we might define the concept `CHARDONNAY-WINE` as a wine whose grapes include chardonnay; the restriction would be written as

(**FILLS** `grape Chardonnay`).

A **SAME-AS** restriction requires that the individual found by following one attribute-path is the same individual as that found by following a second attribute-path. For example, suppose that there is a food and a drink associated with each course at a meal. Then the concept `REGIONAL-COURSE` might be defined as a course where the food's region is the same as the drink's region. This would be written as

(**AND** `MEAL-COURSE` (**SAME-AS** `(food region) (drink region)`)).

2.1.3 Other Restrictions

Tests: There are two operators that allow procedures to be used in specifying concepts: one is used in *CLASSIC* concepts (**TEST-C**), and one is used in *HOST* concepts (**TEST-H**).[7] A test restriction requires that an individual must pass the test to satisfy the restriction. For example, the concept `EVEN-INTEGER` might be defined as the conjunction of the built-in concept `INTEGER` and a test to see if the integer is an even number:

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

(assuming **evenp** is a function in the host language). The individual currently being tested is assumed to be the first argument to the function, and other arguments can be specified as well. Since *CLASSIC* individuals may change, the test functions return one of three values when applied to a *CLASSIC* individual:

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

[7]There are two different operators for tests in order to allow CLASSIC to recognize the realm of any concept directly from its expression. While it can do this with all other constructs, since tests are opaque, CLASSIC can not tell just by looking whether an unmarked test concept is a *CLASSIC* concept or a *HOST* concept. Thus we have two operators, which directly indicate the realm.

- ?: unknown, i.e., the individual is currently consistent with the restriction, but if information is added to the individual, the individual may become either inconsistent with or provably described by the restriction. In other words, the individual neither provably satisfies the restriction nor provably falsifies it;

- T: the individual definitely passes the test, i.e., it provably satisfies it.

Test functions must be monotonic; that is, it should not be possible for the same test function to return T (or NIL) for an individual at one time and NIL (T) at a later time, unless an explicit retraction (see Section 2.3) has been done in between.

Enumerated concepts: A **ONE-OF** concept (or enumerated concept) enumerates a set of individuals, which are the only instances of the concept. For example, a wine whose body could be either full or medium would have the restriction

(**ALL** body (**ONE-OF** Full Medium)).

Primitive concepts: Normally, when one gives a CLASSIC definition for a concept, it is both necessary and sufficient. For example, if we define a FULL-BODIED-WHITE-WINE as a FULL-BODIED-WINE and a WHITE-WINE, we expect the relationship to be an "if and only if" relationship. The **PRIMITIVE** and **DISJOINT-PRIMITIVE** operators allow a user to form concepts that cannot be fully specified by necessary and sufficient conditions. These operators can only define concepts in the *CLASSIC* realm. If we want to define a wine as a drink with special properties we do not want to or cannot fully specify, we would define the concept WINE as (**PRIMITIVE** POTABLE-LIQUID *wine*), with *wine* being an arbitrary symbol (the *index*) used simply to distinguish this concept from others.[8] WINE is then known to be different from any other **PRIMITIVE** concepts defined under POTABLE-LIQUID (i.e., those with different indices—see the discussion on indices in Section 4.1). A **DISJOINT-PRIMITIVE** concept is just like a **PRIMITIVE** concept, except that any concepts within the same "disjoint grouping" are known to be disjoint from each other, and thus, no individual can be described by two **DISJOINT-PRIMITIVE**s in the same disjoint grouping. For example, if we know that fish and shellfish are both types of seafood, and nothing can be both a fish and a shellfish, then we could define fish and shellfish as disjoint primitives under seafood within the same disjoint grouping. That is, we would define the concept FISH as (**DISJOINT-PRIMITIVE** SEAFOOD *type* *fish*), and we would define the concept SHELLFISH as (**DISJOINT-PRIMITIVE** SEAFOOD *type* *shellfish*), where *type* is an arbitrary symbol designating the grouping.

2.1.4 Rules

Aside from the language constructs used in forming concept and individual expressions, CLASSIC allows for forward-chaining rules. A CLASSIC *rule* consists of an antecedent and a consequent, both of which are concepts, where the antecedent must be named. As soon as an individual is known to satisfy the antecedent concept, the rule is "triggered," and the individual is also known to satisfy the consequent concept. For example, if there is a rule that says that the best wine for a dessert course is a full-bodied, sweet wine, then if Mary is eating a dessert course, the rule is fired and CLASSIC will deduce that her course

[8] Any symbol at all can be used as an index. We use symbols that mirror the names of the concepts just to make it easier to keep them straight. There is absolutely nothing special about the symbol "*wine*."

is one whose wine is full-bodied and sweet. Consequents of rules are treated as derived information—if the antecedent of a rule is retracted from an individual, then the consequent is also retracted (see Section 2.3). (This differs from the treatment of rules in typical rule-based systems, such as OPS, where the consequents of retracted antecedents remain in the knowledge base.)

2.2 Knowledge Base Inferences

CLASSIC provides a number of different deductive inferences. The three main types are completion, classification/subsumption, and rule application. *Completion* involves computing the implicit logical consequences of assertions about individuals and descriptions of concepts. For example, when a new concept is defined in terms of existing concepts, *inheritance* is used to determine all of the properties of the new concept—the new concept "inherits" all of the properties from the existing concepts. Thus, if `WINE` is defined to have exactly one body, flavor, and color, and `WHITE-WINE` is defined as a `WINE` whose color is white, then `WHITE-WINE` will inherit the properties that it has exactly one body, flavor, and color, in addition to having the color white. When a new individual is described in terms of existing concepts, it inherits the properties of those concepts. For example, if `Chateau-d-Yquem-Sauterne` is an individual which is, among other things, a `WHITE-WINE`, then it inherits from `WHITE-WINE` the property that its color is white. It also inherits from `WINE` the properties that it has exactly one body, flavor, and color.

When a new concept or individual is created, all of its properties are *combined*, which can lead to a number of conclusions. Suppose that the concept `FULL-OR-MEDIUM-BODIED-WINE` is defined as a wine whose body is either full or medium:

(**AND** `WINE` (**ALL** `body` (**ONE-OF** `Full Medium`))),

and the concept `MEDIUM-OR-LIGHT-BODIED-WINE` is defined as a wine whose body is either medium or light:

(**AND** `WINE` (**ALL** `body` (**ONE-OF** `Medium Light`))).

Suppose that we define the concept `SPECIAL-BODIED-WINE` as both a `FULL-OR-MEDIUM-BODIED-WINE` and a `MEDIUM-OR-LIGHT-BODIED-WINE`:

(**AND** `FULL-OR-MEDIUM-BODIED-WINE MEDIUM-OR-LIGHT-BODIED-WINE`).

CLASSIC combines the properties inherited on the `body` role by intersecting the two **ONE-OF** restrictions, and discovers that the `body` for `SPECIAL-BODIED-WINE` must be `Medium`.

As another example, suppose that Mary wants to serve a regional course (the food and drink are from the same region). She is not very knowledgeable about regions of wines, but she would like to serve a Chianti wine. She knows it is from either France or Italy, so she decides to serve either beef bourgogne or lasagna—whichever one is consistent. She attempts to create an individual course with the following definition (note: `CHIANTI` is considered a general class here, `Beef-Bourgogne` an individual food):

```
(AND REGIONAL-COURSE
     (ALL drink CHIANTI)
     (FILLS food Beef-Bourgogne)).
```

CLASSIC will not accept this course description, because the food and drink are from different regions. If Mary were instead to create the course description with the `food` being `Lasagna`, the assertion would be successful.

When combining properties of an individual, CLASSIC may discover that a role is *closed*, i.e., it can have no more fillers. For example, suppose a wine is defined to have exactly one maker, which is a winery. If the individual `Kalin-Cellars-Semillon` is known to be a wine with maker `Kalin-Cellars`, perhaps represented as

(**AND** `WINE` (**FILLS** `maker Kalin-Cellars`)),

then the `maker` role is implicitly closed by CLASSIC on `Kalin-Cellars-Semillon`, since it can have no more fillers. Thus, if the user tries to add a filler to the `maker` role, this will cause an error. The user may also explicitly close a role (see Section 2.3).

When a new individual is created, inheritance and combination of properties may also cause certain information to be *propagated* to another individual. For example, suppose we know that Sue drinks Chateau d'Yquem Sauterne, and we tell CLASSIC that Sue drinks only dry wines. The information is then propagated that the individual `Chateau-d-Yquem-Sauterne` must be a dry wine. *Contradiction detection* will take place during propagation of properties. In this example, if Chateau d'Yquem Sauterne were already known to be a sweet wine, a contradiction would be detected. When a contradiction is found on an individual, the assertion that caused the contradiction is retracted (i.e., that Sue drinks only dry wines), and all the inferences done up to the point of discovering the contradiction are undone (`Chateau-d-Yquem-Sauterne` is reverted back to being a sweet wine).

When a new concept is defined, and all of its properties are inherited and combined, CLASSIC determines whether the concept is *incoherent* (i.e., if the concept can have no instances because it contains inconsistent information). For example, if the concept `FULL-BODIED-WINE` is a wine whose body must be full, `MEDIUM-BODIED-WINE` is a wine whose body must be medium, and a wine must have exactly one body, then

(**AND** `FULL-BODIED-WINE MEDIUM-BODIED-WINE`)

will be detected to be an incoherent concept, since a wine cannot have a body of both full and medium at the same time—it cannot have more than one body.

When a new concept is defined, *classification* is used to find all concepts more general than the new concept and all concepts more specific than it. For example, suppose that the concept `FULL-BODIED-WHITE-WINE` is defined as a `WINE` whose `body` is `Full` and whose `color` is `White`. When it is classified, the concepts `FULL-BODIED-WINE` and `WHITE-WINE` would be found as parent (more general) concepts (assuming these concepts have been previously defined), while the concept `FULL-BODIED-STRONG-WHITE-WINE` would be found as a child (more specific) concept (assuming it has been previously defined). During classification, *subsumption* is used to determine whether one concept is more general than another concept. In this example, `FULL-BODIED-WINE` would be found to *subsume* `FULL-BODIED-WHITE-WINE`, since it is impossible to have an instance of the latter that is not an instance of the former. Rules are ignored when determining whether one concept subsumes another.

When a new individual is created, classification is also invoked, to find all concepts that are satisfied by the individual. For example, suppose that the individual `Forman-Chardonnay` is known to be a `WINE` whose `body` is `Full`, whose `color` is `White`, and whose flavor is Moderate. When it is classified, it would satisfy the concept FULL-BODIED-WHITE-WINE, but not the concept FULL-BODIED-STRONG-WHITE-WINE. When a new concept with a test restriction is defined, and a subsumption test is done between that concept and another existing concept, also containing a test restriction, the COMMON LISP functions are not analyzed to see if one is more general than another. However, when a new individual is created, and a check is done to see if that individual satisfies an existing concept containing a test restriction, the test function is run on the individual to see if the individual satisfies the restriction.

As discussed in Section 2.1.4, a CLASSIC rule consists of an antecedent and a consequent, both of which are concepts. When an individual is known to satisfy the antecedent concept of a rule, the *rule is applied*, or "triggered," and the individual is also known to satisfy the consequent concept. In the example from Section 2.1.4, when Mary is known to be eating a dessert course, the rule is fired that asserts that the wine she drinks with the course is a full-bodied, sweet wine. If she is known to be drinking a dry wine, then a contradiction is signaled, because the information implied about the wine she is drinking is inconsistent.

2.3 Knowledge Base Operations

There are a number of operations a user can perform on a knowledge base in CLASSIC. The user can *query* the knowledge base for information, by asking the following types of questions:

- "What are all the instances of this concept?" ("Which individuals satisfy this description?")
- "Which concepts does this individual satisfy?"
- "Which individuals fill role r on individual I?"
- "How is role r restricted on concept C (or on individual I)?"

A user can *define* a new concept, role, or individual. This may cause any of a number of inferences to be performed (see Section 2.2). A user can also *add* information to a known individual. For example, if the user originally asserts that Mary has exactly one child, she might later assert that Mary's child is Sue. Concept definitions cannot be modified, although a user can add new rules with any concept as an antecedent at any time.

A user can assert about an individual that a specific role is *closed*, i.e., its current fillers are the only fillers (unless a role is closed, explicitly with a function call, or implicitly when the number of fillers reaches the **AT-MOST** restriction, it may have more fillers, since there is no closed-world assumption in CLASSIC—see Section 4.7). There is no **CLOSE** operator in the expression language. Instead, there is a separate function used to close a role on an individual.[9]

Information that has previously been asserted about an individual can be *retracted* in CLASSIC. For example, suppose Mary was originally defined to be a PERSON, and then she is asserted to be a NON-WINE-DRINKER (a person who drinks no wines). If someone then sees Mary drinking wine, he or she could retract the information that Mary is a

[9]This is because a **CLOSE** operator would provide a different kind of knowledge (autoepistemic) from all other operators.

NON-WINE-DRINKER. In that case, Mary would revert back to being simply a PERSON, and any inferences that may have been made due to her being a NON-WINE-DRINKER are undone. The user can also retract rules that have been added to the knowledge base. No other information about concepts can be changed.

3 When is CLASSIC Appropriate?

As we have seen, CLASSIC includes both a language for representing certain kinds of knowledge, and a system that supports the manipulation of descriptions in this language. As such, it is part of a large family of computer systems variously known as data or knowledge base management systems. As with all such systems, CLASSIC has certain characteristics that make it appropriate for some applications and inappropriate for others. These key characteristics include the following:

- *object-centered*: all individuals have a unique, intrinsic and immutable identity obtained at time of creation; the user cannot form arbitrary logical sentences;
- *terminological*: the system supports the definition of complex "noun phrases" in the form of concepts (and the discovery of their inter-relationships); these concepts can then be used to make assertions about objects. CLASSIC is therefore good at describing complex objects, but not particularly suitable for making complex assertions, such as ones involving multiple quantifiers or disjunction;
- *deductive*: CLASSIC is not just a passive repository for unconnected assertions, like a relational database; the system actively searches to find an entire class of propositions entailed by the facts it has been explicitly told;
- *incremental*: partial, incomplete descriptions of individuals are acceptable;
- *supports knowledge retraction*: the system tracks dependencies between facts and allows certain facts to be retracted;
- *supports simple rules*: these are applied in a simple forward chaining manner, whenever appropriate individuals are found;
- *supports procedural tests*: complex concepts, not otherwise expressible in CLASSIC, can be described procedurally in the host language, so that individuals satisfying them can be recognized;
- *well-integrated with the host language*: CLASSIC allows values from the host programming language to be managed as instances of their own classes without requiring them to be "encoded" as CLASSIC individuals.

These characteristics allow CLASSIC to provide a great deal of power for certain types of applications, but also limit its utility in some situations.

3.1 When to Use CLASSIC

The most notable feature of CLASSIC's family of languages is the "self-organization" of the concepts defined: because concepts have clear definitions, it is possible to have the *system* organize them into the subsumption hierarchy, rather than have the user specify their exact place. This is important because standard logic and production systems, for example, do not address the knowledge engineering issue of organizing large collections of knowledge. Thus, CLASSIC, and more generally, its "sibling" languages can be exploited in any domain where it is useful to organize a large set of objects that can naturally be represented in terms of "features" or "roles." For example, it has been argued that this kind of automatic classification is a useful way of organizing a large set of rules in an expert system [Yen *et al.*, 1989]: by classifying the left-hand sides, the system automatically calculates a well-founded specificity ordering over the rules (the generalization hierarchy); this can be used directly in conflict resolution.

Another example of such a family of applications would be information retrieval, where every object[10] has a complex description, and a query may be phrased as a description of objects having a certain structure (e.g., "find all meals with at least two courses, each of which has a sweet wine as its drink"). In such cases, the descriptions can be classified with respect to each other so that similar objects are grouped together. This can provide a much more sophisticated indexing scheme than simple keyword schemes, without increasing retrieval time significantly since everything is preclassified. (The cost for this type of system is at concept classification time, but presumably that would not be a problem in a library scenario.) The LASSIE system [Devanbu *et al.*, 1990; 1989] is one example of such an application: it maintains information about a large software system and its components, viewed from multiple perspectives, and it can be queried as part of the effort of understanding the software system. LASSIE accepts queries in the form of structured object descriptions (e.g., "an action that drops a user from a call and is caused by a button-push by an attendant"), and uses classification to find all matching instances of the query. LASSIE was first implemented in the KANDOR language, and has now been converted to CLASSIC.

Because the hierarchy of concepts can change dynamically, CLASSIC and its close relatives are also more appropriate for database-like applications that have an *evolving schema*—the normal state of affairs in design and specification efforts, for example. In contrast, standard database management systems are relatively poor at supporting schema changes, in comparison to straight updates to data.

Another important class of applications consists of those involving incrementally evolving descriptions. In contrast to standard repositories of data, such as traditional databases, a CLASSIC knowledge base allows the user to maintain a partial, incomplete view of the domain of discourse, a view in which information is incrementally acquired. The following are some of the features of CLASSIC that support this:

- role fillers of individuals can be described in ways other than by simple enumeration; for example, it is possible to
  - assert how many objects an individual is related to via some role, without knowing the actual objects (e.g., "every wine has at least one object related to it via the `grape` role");
  - describe the fillers of a role, without knowing them; for example, "all the fillers of the `drink` role for this course are from France";
- incomplete information may be gradually refined as new knowledge is acquired; thus
  - a particular meal can be said to have at least three courses, and then later discovered to have at least four;

[10] An object might be a text document, some software component, a chemical compound, a meal, etc.

  - a particular individual may first be known to be an instance of FRUIT (some primitive class), and then later be discovered to be an instance of GRAPE (a more specialized primitive class), without knowing the exact variety of grape (each of which is a primitive subclass of GRAPE);

- the "closed world assumption," normally invoked in data and knowledge bases, views the state of knowledge to be complete at any time; therefore when additional information (not contradicting past data) is added, one is often faced with the problem of having to retract certain conclusions that were reached "too hastily." The absence of the closed world assumption in CLASSIC avoids these problems by not drawing conclusions until all information is known, and hence CLASSIC supports incremental filling-in of a partially-known situation.

This ability to handle partial knowledge can be usefully exploited in such tasks as the design or configuration of artifacts (where something is being created, without having an exact idea of all its parts until it is completed), or the "detective" process involved in recognizing objects from clues discovered over time (e.g., identifying criminals). Languages in the KL-ONE family have been used for such purposes in configuration tasks [Owsnicki-Klewe, 1988], among others.

CLASSIC is also suitable for applications that want to enforce constraints on collections of facts because inheritance is strict and "trigger"-like rules are available. We have one application (a configurator) that uses CLASSIC mostly as an integrity checker. This application makes use of inheritance by putting constraints on high level concepts and then lets CLASSIC enforce the constraints on all subconcepts, avoiding the redundancy that would be necessary in many database implementations of the same facts.

CLASSIC, unlike other languages of its kind, has been designed to allow the relatively easy integration of individuals from the host programming language in a manner consistent with *CLASSIC* individuals. This makes CLASSIC easier to use in situations where values such as integers, etc., need to be stored in the knowledge base, and in the case of languages like COMMON LISP, it allows arbitrary data structures and programs to be kept in a CLASSIC knowledge base—an important feature for AI applications to Software Engineering, for example.

Because of the object-centered nature of CLASSIC, individuals can be created without knowing some or all of their final descriptors. This allows a user to take the following set of steps: 1) create some new "dummy" individual; 2) relate it to some existing individual (e.g., as a role filler); and 3) inspect the KB to see what additional descriptors have been attached to the dummy individual as a result of rule firings and other deductions. The result is a technique for obtaining so-called "intensional" answers to queries—descriptions of conditions that must hold of *any* individual, currently existing or not, which satisfies certain relationships (see [Borgida *et al.*, 1989] and Section 6.7 for more details). Such querying is not supported by traditional databases.

3.2 When Not to Use CLASSIC

Previous sections have mentioned the goals and philosophy behind the design of CLASSIC. In keeping with our principles of providing effective reasoning services, certain expressive features have been deliberately left out of the language. These features obviously influence the situations where CLASSIC is appropriate as a representation tool.

Because of its object-centered nature, CLASSIC is likely to be cumbersome to use in cases where mathematical entities such as tuples, sequences, geometric entities, etc., are the center of attention. This is because such entities usually have a notion of "equality" based on (recursive) component identity. For example, calendar dates are structured objects, and it seems natural to model them as CLASSIC individuals with three attributes: `day`, `month`, and `year`. However, object identity may provide surprising results: if we are tracking the date on which wines are bottled through an attribute `bottled-on`, and we are interested in finding out whether two bottles `Wine-bottle-53` and `Wine-bottle-661` were bottled the same day, then simply checking that `Wine-bottle-53`'s `bottled-on` is the same as `Wine-bottle-661`'s `bottled-on` may result in the answer "false" even if the two dates have the same `day`, `month`, and `year`. In order to avoid such problems, the user would have to search the knowledge base before entering any date, to make sure that a date with the same attribute values did not already exist.[11]

With CLASSIC, an application requiring simple retrieval of told facts, with no interest in derived consequences or a complex query language, will pay an unnecessary performance penalty (both in time and in space) during the processing of input data, and especially in the revision of told facts, since updates would normally be quite simple in that case. Furthermore, at least at the moment, CLASSIC does not have efficient data access facilities built-in in order to handle very large numbers of individuals, such as desired in data-processing applications.

Since CLASSIC does strict inheritance, defaults and exceptions are not easily encoded in the language. If an application is inherently oriented toward defaults, CLASSIC should not be the language of choice. If, however, there are only a small number of certain kinds of defaults, CLASSIC may be adequate (see Section 6.3).

CLASSIC provides only a limited form of rules, where both the antecedent and the consequent refer to the membership of a *single* individual in some concept (which of course might be structured). Applications requiring complex conditions in the antecedent are much more difficult to handle properly. First, CLASSIC supports neither full negation nor full disjunction, so these constructs are not usually available for expressing complex trigger conditions (but see Sections 6.1 and 6.2). Nor is it possible to write rules that are triggered by the existence of two or more individuals that are not directly related by some chain of roles (e.g., "if there exist wines x and y such that one is twice as old as the other, then…"). One could consider using something like OPS5 as a front-end rule-processing system and use CLASSIC as a back-end structured working memory. An alternative explored in [Yen *et al.*, 1989] has been to expand the role of the knowledge base to manage both the space of rules and the policy of rule firing.

CLASSIC does not have full negation. If an application will constantly need to refer to a concept that includes everything that is not an instance of some other concept, then the application is not well-suited for CLASSIC. Limited uses of negation are discussed in Section 6.1.

Classification systems such as CLASSIC are usually implemented as forward-chaining

[11] In CLASSIC, this problem could sometimes be resolved through the use of complex objects in the host-language domain, as long as the host language performs equality checking in a component-wise fashion on certain data structures, such as is the case with COMMON LISP's `equal` predicate. However, in that case, the internal structure of the objects of interest (e.g., dates) would not be accessible to CLASSIC for reasoning.

inference systems. (By way of contrast, queries in PROLOG and databases augmented with recursive rules are usually processed by working backward from the query to the database of explicitly asserted facts.) This means that the addition of new concepts or individuals is time-consuming, though retrieval is more efficient. Therefore if updates are frequent and time-critical, current implementations would make such systems less than ideal when the number of objects becomes large.

Because CLASSIC distinguishes individuals from (generic) concepts, and does not support "meta-concepts," CLASSIC itself is not suitable in situations where some individual may in certain cases be viewed as a class with instances. For example, there is no direct way to associate with the concept `WINE` a specific value through a role such as `average-age` or `maximum-sugar-content`—roles that do not make sense when applied to individual bottles of wine. Note however that this is not an intrinsic lack of KL-ONE-style languages—it could easily be remedied in future generations.

Similar problems arise in situations where the "ontology" of the domain is not self-evident: in a knowledge base about wines, does an instance `Kalin-Cellars-Chardonnay` of the concept `CHARDONNAY-WINE` correspond to a specific kind of wine, to a particular vintage ("the 1985 one"), or, even more specifically, to a particular bottle? In the case of the vintage, is it after bottling, or later on, or both? Such shifts of perspective are not easily supported by knowledge representation languages that maintain a strict distinction between individuals and concepts (see Section 5.1.1).

Finally, CLASSIC and its relatives have general (weak) reasoning procedures, and do not support the direct and efficient addition of specialized kinds of inferences. This means that applications needing to make intensive use of temporal reasoning or spatial reasoning, for example, would find it difficult to have CLASSIC deduce the desired relationships (but see [Litman and Devanbu, 1990] for an extension to CLASSIC that makes it more useful in planning applications).

While some of the above limitations are inherent to the object-centered view of CLASSIC, extensions to the system may eventually relax some of the other restrictions. Under active consideration now are the addition of defaults, a more elaborate rule framework, and large-scale data storage facilities with a powerful query language.

4 Difficult Ideas

Once you have decided to use CLASSIC to build a knowledge base it is important to understand several subtle issues. We will address these in relation to CLASSIC; however, many are equally applicable to the other languages in the KL-ONE family. The issues concern the philosophy of the language and knowledge-base design, and can affect decisions concerning the gross structure of the KB. The issues include the amount and kind of information that should go into a concept definition, individuals versus concepts, CLASSIC's detection of incoherencies in role fillers, when rule application occurs, how CLASSIC handles unknown individuals, how updates are done, and the impact of eschewing a closed world assumption. Two other key (and somewhat difficult) ontological considerations are covered in Section 5.1.

4.1 Primitive and Defined Concepts

It has been traditional in the KL-ONE family of languages to provide for two kinds of concepts—defined and primitive. A defined concept is like a necessary "if and only if" statement in logic. For example, if a white wine is defined to be exactly a wine whose color is white, then deductions can be done in two directions:

- if we know something is a white wine, then we know that it is a wine and it is white;
- if we know something is a wine and has color white, then we know it is a white wine.

In other words, this kind of definition includes necessary and sufficient conditions for membership in the class. So, if `WHITE-WINE` is defined in the obvious way, any object that is asserted to be one will be both a wine and something whose color is white; also, anything that is known to be a wine and have white color will be classified as a `WHITE-WINE`.

A primitive concept includes only necessary (but not sufficient) conditions for membership. In contrast to defined concepts, primitive concepts support deductions in only one direction (like an "if" statement instead of an "if and only if" statement). For example, it is hard to define "wine" completely. So one might say that, *among other things*, a wine is something that has a `color` that is either `Red`, `White`, or `Rose`. In this case, when CLASSIC is told that something is a wine, it will infer that it has a value restriction on the `color` role, but just because something has a `color` role filled with value `Red`, CLASSIC does *not* infer it to be a wine.

Determining whether a concept should be primitive or defined is a key aspect of building a CLASSIC KB. The basic idea is that a primitive concept is appropriate when no complete definition exists or when only part of a completely known definition is relevant. In the former case, we have no choice but to use a primitive concept—if we use a defined concept, accidental and inappropriate "only if" deductions will be sanctioned. In the latter case, there may be no need to bother with a complete definition if the application never demands that the system automatically recognize an instance of the concept. If the user can be guaranteed to assert class membership directly, then a full definition of a concept like `WINE` is not necessary, even if one is possible. Defined concepts are appropriate when the complete definition is known and relevant, or when one wants the *system* to determine membership in a class. Primitive concepts are usually found near the top of a generalization hierarchy and defined concepts typically appear as we move further down by specializing general concepts with various restrictions.

In CLASSIC, primitive concepts are distinguished by indices. Thus concepts `FOOD` and `WINE` could be defined as CLASSIC terms (**PRIMITIVE** `CLASSIC-THING *food*`) and (**PRIMITIVE** `CLASSIC-THING *wine*`) respectively; the indices `*food*` and `*wine*` allow these two concepts to be different, and at the same time permit synonyms to be defined: `FAVORITE-BEVERAGE` might also be defined as (**PRIMITIVE** `CLASSIC-THING *wine*`). The use of indices reinforces that the meaning of a primitive concept definition is contained in its expression—as is the case with all other CLASSIC descriptor types—while the name is simply a label that helps the user. Defined concepts do not need an index as they are distinguished from other concepts by their very definitions. Synonyms can also be created by defining two concepts with equivalent descriptions. Both the concept names may be used later, but in the concept hierarchy they refer to the same entity.

In general, there are three reasons to consider creating a defined concept in systems like CLASSIC:

1. The most important reason is simply that the meaning of an important domain term can be fully defined within the language. In many cases, there will be a natural name in the domain for the concept and an obvious set of necessary and sufficient conditions. For example, `OENOLOGIST` might be defined as a `PERSON` who studies wines. There will be many of these concepts in an artificial domain, and few if the domain covers mainly naturally occurring objects.

2. In some ontologies, it can be useful to organize the antecedents of rules into a taxonomy. Rules can be organized so that each consequent is associated with an antecedent at the right level of generality, and rules that apply to more general situations can be inherited and applied in specific situations. This allows classification—and not just direct assertion—to determine when a rule is invoked. For example, as in our sample knowledge base (see Section 5.3), we might have partial knowledge about an appropriate wine associated with the general property that a course's food is seafood (i.e., the wine's color must be white), and another fact associated with a more specific property, for example, that the course's food is shellfish (i.e., the wine must be full-bodied). Organization of the antecedents into a hierarchy makes the ontology clearer and makes knowledge base maintenance substantially easier. Here a defined concept is simply used to express the antecedent of a rule, and need not correspond to any natural class in the domain; such concepts will most likely not have any naturally-occurring names in the domain. In our sample KB, we have used constructed names like "`SHELLFISH-COURSE`" for these concepts, although such names hold no significance other than as placeholders (the antecedents of rules in CLASSIC must be named).

3. For some primitive concepts, there may be a number of ways that class members can be recognized, even if there is not a single necessary and sufficient definition. A final use for defined concepts is to express sufficiency conditions for recognition of members of an otherwise primitive class. For example, while `PERSON` would most likely be primitive in most ontologies, conditions like "featherless biped" and "child of a person" might be considered sufficient conditions for determining personhood. In CLASSIC, one can use a defined concept to represent each set of sufficient conditions

(e.g., FEATHERLESS-BIPED would be a defined concept). Each such concept would be the antecedent of a rule whose consequent was the primitive concept whose members were to be recognized (PERSON, in this case).

4.2 Definitional and Incidental Properties

It is important in CLASSIC to distinguish between a concept's true definition and any incidental properties that its instances all share. For example, consider red Bordeaux wines, which are always dry. The color and the region would clearly be part of the definition of the concept RED-BORDEAUX-WINE, since this constitutes part of the very meaning of the term. But the property of being dry is certainly not part of the meaning of "red Bordeaux wine," even if it is a (contingent) universal property of red Bordeaux. Thus, in a CLASSIC-style representation the first two properties, (**FILLS** color Red) and (**FILLS** region Bordeaux), would be part of the concept RED-BORDEAUX-WINE, whereas the third would be expressed as a rule, whose consequent would be (**FILLS** sugar Dry) and whose antecedent would be RED-BORDEAUX-WINE.

The distinction between definitions and incidental properties is not important in KR systems that do not perform classification, as it has no effect on how these systems work. However, in CLASSIC, since they represent only necessary, and not sufficient conditions, rules do not participate in either recognition or classification. So, for example, putting the "dryness" property into the definition of RED-BORDEAUX-WINE would mean that a wine would have to be dry to be recognized as a RED-BORDEAUX-WINE (as opposed to having "dryness" automatically asserted about wines that have already been recognized as RED-BORDEAUX-WINEs); it would also mean that RED-BORDEAUX-WINE would be inappropriately classified under the concept DRY-WINE. (See also Section 5.3, especially footnote 19.)

This type of inappropriate classification also affects primitive concepts. Consider the earlier primitive definition of WINE as something that has, among other things, a color that is either Red, White, or Rose:

```
(PRIMITIVE (AND (ALL color (ONE-OF Red White Rose))
                (AT-LEAST 1 color))
           *wine*).
```

Another way to view this might be to make WINE an atomic primitive concept (i.e., directly below CLASSIC-THING), and use a rule to express the color restriction. In both cases, since WINE is primitive, the color restriction would not be used to answer subsumption questions. Also, if an individual were stated to be a wine, in both cases, the individual's color role would be checked for consistency with the restriction. However, there is an important difference. If we added a defined concept, COLORED-THING (something that has at least one color), then if WINE were only a primitive thing that had a color restriction in a rule, it would not be classified under COLORED-THING. The WINE concept that included the restriction as part of its meaning would, on the other hand, get classified under COLORED-THING.

The distinction between definitional and incidental properties must be carefully made for *all* concepts in CLASSIC, not just defined concepts. In general, the user must decide on ontological grounds whether a restriction should be taken as part of the *meaning* of a concept (and thus participate in classification and recognition) or simply as a derived property to be inferred once class membership is ascertained. The difference between primitive and defined concepts is that in the former case class membership must be asserted directly (by the user or a rule), and in the latter the system can determine it.

4.3 Concepts and Individuals

Although in some ways concepts look very similar to individuals (e.g., CLASSIC's syntax allows the same types of expression for each), there are some subtle (and some not so subtle) differences between them. It is useful to understand some of the important distinctions when trying to understand CLASSIC's classification and deductive processes. First, individuals have unique identities and are countable. An individual can be described by concept expressions that apply to it, but there is a uniqueness assumption that guarantees that two individuals with different names—even with the same description—will be different individuals. Concepts are descriptions and because of the compositional nature of descriptions, the concept space is infinite. The concept hierarchy could include things like full-bodied-wines, full-bodied-white-wines, full-bodied-white-medium-flavored-wines, etc. When considering the knowledge base, it makes sense to count the individual wines but it is not clear how or why one would want to count all the descriptions of those wines.

Next, facts in the world can change, and thus individuals can change, too. One might want to add information to a particular individual or perhaps change something about it, for example, the price of a wine. In contrast, concept definitions and their relationships to each other do not change. Once someone defines a white wine, say as a wine whose color is white, CLASSIC will continue to classify all individuals and concepts with respect to this definition until someone reloads the entire knowledge base. A more subtle issue is that retraction and addition of facts about individuals do not change the concept classification hierarchy. Individuals, and their classification, can change through assertion and retraction of facts; but the semantics of CLASSIC was designed to make the concept hierarchy be immune to changes in individuals. (The concept hierarchy would change monotonically if a new concept definition were added.)

For example, given a concept `PICNIC-BASKET` defined as

```
(AND BASKET
     (AT-LEAST 2 drink) (AT-MOST 2 drink) (ALL drink WINE)
     (AT-LEAST 3 food) (ALL food EDIBLE-THING)),
```

a `CALIFORNIA-PICNIC-BASKET` defined as

```
(AND PICNIC-BASKET (ALL drink CALIFORNIA-MADE)),
```

and a `KALIN-CELLARS-BASKET` defined as

```
(AND PICNIC-BASKET
     (FILLS drink Kalin-Cellars-Chardonnay Kalin-Cellars-Cabernet)),
```

then even though both the wines in the definition of `KALIN-CELLARS-BASKET` happen to be made in California, `KALIN-CELLARS-BASKET` will be classified under `PICNIC-BASKET` but *not* under `CALIFORNIA-BASKET`. The motivation is that the concept hierarchy should not have to change if the incidental facts about one individual changed. If Kalin Cellars moved its winery to Oregon, we would not want to have to reclassify the concept `KALIN-CELLARS-BASKET`. Note, however, that if there were an individual `Kalin-Cellars-Basket-1` that was a

KALIN-CELLARS-BASKET, this would in fact be classified under CALIFORNIA-BASKET. The difference is that this is an *individual*, and as such it is classified based on the known properties of all individuals, including its role fillers. Concepts are not classified based on properties of individuals; they are only classified based on information that is *necessarily* true. The individual Kalin-Cellars-Basket-1 could later be reclassified if the properties of either Kalin-Cellars-Chardonnay or Kalin-Cellars-Cabernet were changed or modified.

As mentioned previously, in CLASSIC, rules function differently with respect to concepts and individuals. Rules are associated with concepts but they are not "fired" until an individual is found to be an instance of the concept. Thus, although there may be a rule that says that wines for seafood courses must be white, this rule would not be enforced until there was a known individual seafood course.

4.4 Rule Application

A rule (see also Sections 2.1.4 and 6.4) is not actually "fired" until an individual is found to be an instance of the antecedent concept. Thus, if one creates a rule that says that white wines must be drunk with seafood courses, this information does not get propagated until a seafood course exists. One ramification of this is that in order to test all the rules in a knowledge base, e.g., for global consistency, one needs to create individual instances of all the concepts that are the antecedents of rules. For example, consider the SEAFOOD-COURSE concept above and a concept SHELLFISH-COURSE that is a kind of SEAFOOD-COURSE with a rule stating that the wine drunk with a SHELLFISH-COURSE must be a full- or medium-bodied wine. In order to check consistency of the rules and to observe restrictions appearing on the wines of courses, individual seafood and shellfish courses would need to be created. Once we created a shellfish course with an associated wine, we would find that the wine would be restricted to being a white, full- or medium-bodied wine.

Because the right hand sides of rules are concepts and not commands, it is not possible for a retraction to result from the application of a rule. Thus, the only thing that a rule may do is state that if an individual is found to be an instance of the antecedent concept, then it is an instance of the consequent concept. If this is not consistent with the other facts in the knowledge base, then the statement about the individual that triggered the firing of the rule would not be allowed as input to the knowledge base.

It should be noted that rules work in one (and only one) direction. In the previous example, because a course is a seafood course, then we know that the wine for the course must be a white wine. The system would not make the backward inference that because a wine for a course is not a white wine, then the course must not be a seafood course.

4.5 Unknown Individuals in CLASSIC

One of the advantages of CLASSIC, as pointed out earlier, is that it allows the description of partially known objects. For example, one way to give information about "null values"—values that exist but are not currently known to the KB—is through identities between attribute paths. We can say, for example, that the Thanksgiving day menu will have the same drink for lunch as for dinner (by adding **(SAME-AS (lunch drink) (dinner drink))** to the description of Thanksgiving-Day-Menu), without knowing the identity of the lunch, dinner or drink objects.

More usually, it is possible to give filler information about roles of unknown objects; for example, one can take an individual course, `Course-1`, and add to its description the restriction

(**ALL** `drink` (**FILLS** `grape Riesling`))

to state that the wine served with it is made from Riesling grapes, without knowing the actual wine to be served.

These examples might make one believe that the system actually creates and maintains CLASSIC individuals for all entities in the domain implied by the current knowledge base (these are sometimes called “Skolem individuals”). This, however, is not the case. We cannot say that some restaurant's wine list includes the drink of `Course-1`, and then, later on, when we find out what is the specific drink of `Course-1`, expect it to show up on the wine list.

In the current implementation of CLASSIC the processing of individuals is complete only in the case when the fillers of roles are all known. The following two examples illustrate incompleteness that occurs when some role fillers are not known.

First, in order to determine that some individual `Ind` is an instance of a concept of the form (**ALL** `p` (**ALL** `q C`)), it is sufficient to know the complete set of the `q`'s of the `p`'s of `Ind` without necessarily knowing the `p`'s of `Ind`. `Course-1` above illustrates this possibility: we know that the `grape`s of the `drink`s of `Course-1` include `Riesling`, but we don't know the `drink`s; if `Riesling` were known to be a fruit and the `grape` role could have at most one filler, a more complete reasoner would recognize `Course-1` as an instance of the concept

(**ALL** `drink` (**ALL** `grape FRUIT`)).

For the current implementation, we believe that situations in which such conclusions can be reached are sufficiently rare that we have chosen to avoid the ever-present overhead of looking for them.

Second, the implementation does not perform case analysis over the set of possible fillers for some role or role-path. This means that even if `Course-2` has one drink, which is either `Mouton-Cadet` or `Chateau-Lafite`, and both are made in France, the system will fail to recognize that no matter which object is the actual filler of `drink`, `Course-2` should be an instance of the concept

(**ALL** `drink` (**FILLS** `made-in France`)).

We emphasize that the above incompleteness arises only in the presence of the constructions (**ALL** `p` (**FILLS** `q` ...)) and (**ALL** `p` (**ONE-OF** ...)), used because the actual fillers of the `p` role are not yet known. In the current implementation, when information about individuals is incomplete in this way, the subsumption mechanism normally used for concepts is used (since that deals with descriptions intended to be incomplete). However, with that mechanism, the properties of individuals are not considered (e.g., the regions of the wines in the above example; for the reasons for this, see Section 4.3), even though they ought to be when processing individuals.

4.6 Updates

As mentioned in Section 2.3, CLASSIC allows information that has been explicitly asserted by the user about individuals to be retracted. However, CLASSIC does not allow retraction of information that has been *derived* from other information. This is best explained with an example.

Let us begin with an individual that has a restriction on all of the fillers of a role and a known filler for that role, and then try to retract the restriction on the filler. If CLASSIC were told that Lori drinks only kosher wines and that one of the wines that she drinks is `Shalom-Cream-White-Concord`, then `Shalom-Cream-White-Concord` would be inferred to be kosher (by a propagation inference). If at some point, we actually were to discover that `Shalom-Cream-White-Concord` was not kosher, we might want to retract that fact from our knowledge base. CLASSIC would not allow this retraction since its knowledge about this fact is considered to be *derived* information. CLASSIC would force the retraction of some piece of *user-stated* information that led to the conclusion that `Shalom-Cream-White-Concord` was kosher. For example, the user could retract either the fact that Lori drinks `Shalom-Cream-White-Concord` or the fact that Lori drinks only kosher wines.

The reason for disallowing retraction of derived information is to maintain consistency of the knowledge base. If CLASSIC allowed direct retraction of the fact that `Shalom-Cream-White-Concord` was kosher, then if someone asked if it was, it would be unclear how to answer: if the **ALL** restriction on `Lori`'s `drinks` role were enforced, the answer would be "yes"; if the directly stated facts on `Shalom-Cream-White-Concord` were examined, the answer would be "no." Also, if CLASSIC allowed retraction of derived information, some updates would appear never to have occurred. CLASSIC's approach to updates is to retract the stated information and automatically retract all derived information that was based on that information. Then the system rederives all facts that hold in the new situation. If CLASSIC allowed the retraction of the fact that `Shalom-Cream-White-Concord` was kosher, then following this algorithm, it would have to reclassify `Shalom-Cream-White-Concord`. It would once again find that `Shalom-Cream-White-Concord` was a wine that Lori drank and then it would propagate the restriction that the wine must be kosher. Thus the knowledge base would simply revert back to the previous state wherein `Shalom-Cream-White-Concord` was kosher; the update would appear never to have occurred. The only other way to maintain consistency would be for CLASSIC to retract a piece of information that led to the derived information. In this case, it is not clear which piece of information that should be, thus it seems appropriate to force the user to make the choice.

4.7 No Closed World Assumption

CLASSIC does not work under the closed world assumption (CWA) for individuals, that is, it does *not* assume that anything that it does not know is false. Thus, if some basket were known to have two specific wines in it, CLASSIC would not assume that it had *only* two wines in it—it would deduce only that the basket had *at least* two wines in it. So if this same basket had three things to eat in it and we knew that `PICNIC-BASKET`s by definition had at least three things to eat and at most two wines in them, this basket could not be classified as a `PICNIC-BASKET`. It would only be classified as such when the `drink` role became "closed"—i.e., when CLASSIC was told or it derived that there could be no other fillers for the `drink` role. This example shows that in general an individual cannot be classified under a concept with an **AT-MOST** restriction until the corresponding role is closed. The same is true for concepts with **ALL** restrictions.[12]

A role can be closed in two ways. The user may explicitly tell CLASSIC that a particular role on an individual will have no more fillers. Alternately, the system may derive that a role must be closed. If the system is told that an individual is an instance of a `PICNIC-BASKET`, and it also knows that `PICNIC-BASKET` contains the wines `Kalin-Cellars-Chardonnay` and `Marietta-Zinfandel`, then CLASSIC can deduce that the role is closed since the definition of `PICNIC-BASKET` states that there may be at most two wines.

[12] There is one way to classify an individual with respect to concepts with **AT-MOST** or **ALL** restrictions. If the individual in question has a restriction (either directly or by inheritance), then CLASSIC can make deductions based on this restriction, including determining that it implies the target **AT-MOST** or **ALL** restriction. If, for example, CLASSIC is trying to classify something as a `CALIFORNIA-BASKET` and its `drink` role is not closed, but it does have a restriction that all its drinks are made in the Napa Valley, and we know that everything that is made in Napa Valley is made in California, then even without knowing all the fillers of the `drink` role, CLASSIC can make the deduction that it satisfies the **ALL** restriction on `drink` of `CALIFORNIA-BASKET`.

5 Building CLASSIC Knowledge Bases

Once it has been determined that CLASSIC is an appropriate language to use in describing a domain, and some of the more subtle language issues are well in hand, there is still the significant problem of designing the knowledge base given the domain structure. While not identical to the traditional expert systems process, the process of developing a CLASSIC KB is a form of knowledge engineering, where the key is finding the right way to break the domain into objects and their relationships. While there is no single method for producing such an ontology, we discuss some general issues to consider and offer one possible process for creating a knowledge base. We also present parts of a CLASSIC knowledge base, to illustrate the style of description of a typical domain representation.

5.1 Basic Ontological Decisions—Individuals and Roles

Since frame systems like CLASSIC are object-centered, the key idea is to determine what the "objects" in the domain are. This involves the specification of the individual items about which information can be gathered and asserted (the *individuals* of the domain), as well as the specification of classes of those items that share common properties (the *concepts*). The properties of the individuals and the relationships between them are then represented as *roles*. This is all complicated by two key facts: what constitutes an "individual" is not always clear (different levels of abstraction are possible), and some terms seem equally well expressible as concepts and as roles. In all of these cases, the knowledge engineer needs to make a determination fairly early in the KB design process.

Let us consider these two issues in turn, and then we will discuss a general procedure for getting a domain characterized in CLASSIC.

5.1.1 Individuals versus Concepts

Imagine that we are developing a knowledge base of foods and wines. Intuitively, it would seem clear that items like `WINE` and `WHITE-WINE` (a wine whose color is white) should be concepts. It is likewise reasonably clear that `CHARDONNAY-WINE` (a wine made from the chardonnay grape) should also be a concept. However, things are not so simple when we attempt to represent a single "wine."

In some knowledge bases, for example in an application that will recommend a wine to a patron for a general class of dinners (e.g., shellfish), an individual winery's varietal (e.g., Forman Chardonnay) will be an appropriate individual. In our sample knowledge base (Section 5.3), we use this as the level of our individuals. However, for some problems, this level might not be fine-grained enough. For the discriminating wine-drinker, the vintage of a particular wine may be critical, and thus `FORMAN-CHARDONNAY` would have to be a concept, in order that `1981-Forman-Chardonnay` could be an individual. Or, it might be necessary in some applications to make individual bottles of wine be individuals in CLASSIC.

While different kinds of objects can be considered individuals from different points of view, in a system like CLASSIC we are forced to make a commitment at the outset. In that case, the key question to ask is, which objects would be appropriate to *count* in an application? Or, alternatively, in a retrieval application, which objects would be best to retrieve given a query? For a wine-advisory application, the answer given by a wine steward to the question, "How many wines do you stock?" would indicate which items to count as individuals (e.g., `Forman-Chardonnay`). Alternatively, one could count as individuals the items appearing on a menu (e.g., winery, varietal, and vintage).

Whatever level we fix for our individuals, any other descriptions in the domain that could be considered individuals from some other point of view can be handled in one of two less-than-ideal ways. First, they could simply be represented as concepts. Thus, if `1981-Forman-Chardonnay` was an individual, `FORMAN-CHARDONNAY` would be a concept, and the former would probably be described by the latter. An alternative would be to allow both objects to be individuals. But since CLASSIC does not currently support a "meta-description" facility, this representation would be incomplete in an important way, in that CLASSIC would maintain no relationship at all between the two individuals. One could go so far as to place a `generic-varietal` role on `1981-Forman-Chardonnay` and fill that role with `Forman-Chardonnay`, but CLASSIC would treat that role just as any other, and no properties of the more generic varietal individual would be inherited by the more specific vintage one.

5.1.2 Concepts versus Roles

As mentioned, another key distinction that the user of a language like CLASSIC is forced to make is that between concepts and roles. A number of people working with KL-ONE-like languages have reported having difficulty deciding whether something should be a concept or a role. Terms like "father," "landlord," etc., can be used equally well in either sense. For example, "Ron is a new father" uses *father* as a concept. "Ron is the father of Rebecca" uses it as a role. Even a more straightforward term like "grape"—an obvious candidate for concepthood—can present a problem. We can easily imagine the properties of grapes (`color`, `where-grown`, `age-of-vines`, etc.), and can visualize `GRAPE`'s place in a taxonomy of types of foods. However, it is equally plausible to imagine a `grape` role for the concept of `WINE`, indicating the kind of grape a wine is made from. Should *grape* be a concept, a role, or both?

While the treatment of any particular domain term will really depend on the application, there are some general guidelines to use when trying to design concepts and roles. Since part of the problem is the use of nouns in natural languages to correspond to both concepts and roles, we need to look beyond the surface properties of words. In languages like English, certain nouns seem to reflect items that have existence independent of any others (e.g., "person," "apartment," "wine," "grape"), and others reflect items that depend on others for their existence (e.g., "father," "landlord," "vintage," "skin"). The former most obviously correspond to one-place predicates in first-order logic. We would have no trouble describing an individual by one of these terms without reference to any other individuals on whose existence they depend. Thus, we could independently characterize an object as a grape without needing to make reference to any wines made of out of such grapes, nor would there ever have to be any. The description of an item as a grape would stand on its own, without implying the existence of any unmentioned individuals.[13]

On the other hand, while we might naturally use some terms from the latter set as if they were also one-place predicates (e.g., "Deb is a landlord"), *they in actuality imply*

[13]This discussion is intended to be intuitive, and relies only on a naive understanding of the ontology of the world. It is not intended to invoke deep discussion about existence, objecthood, or any other metaphysical issues.

the existence of a second argument (e.g., whom Deb is the landlord *of*). In this case, the primary representation in CLASSIC should be as a role. Any interpretation of the term as a concept would be *derivative* from its interpretation as a role, since there is always an *implied* second argument.

The clear guideline for discrimination between concepts and roles is thus the determination as to whether a description can stand on its own without implying an unmentioned object related to the object in question. In an intuitive ontology, `SHELLFISH` would clearly be a concept, and `vintage` would clearly be a role. There are some cases—including those just mentioned—where it will be quite easy to determine which is which. In the case of an unquestionable concept like `SHELLFISH`, it is almost impossible to imagine using the term in a phrase like, "the shellfish of ⟨something else⟩." That is, it would be very hard to imagine a property of something called its "shellfish." In the case of an unquestionable role like `vintage`, it is almost impossible to consider using the term *without* the "of" phrase. For example, it is unusual to use "vintage" in any other way than as the vintage *of* a particular wine.

Unfortunately, most terms will not be so pure in their natural use. However, the basic guideline still applies. Even though we can refer to a "wine's grape" (i.e., its composition), the concept of a grape stands on its own and does not need to lean on the existence of any wines. Even though someone is referred to as a "father," that description is not truly meaningful without taking into account the implied child. One interesting difference between these two cases (in which a term can be used either as a concept or as a role) is that in the latter case, the value restriction used for the `father` role would have a different name (`MAN`) than the role, whereas it seems most natural in the former case to name the role with the same name as the value restriction (the `grape` role of a `WINE` would be filled by a `GRAPE`). It would seem somewhat silly and uninformative to have the value restriction of the `father` role be `FATHER`. This is because the *only* difference between the concept `MAN` and any proposed concept like `FATHER` is the man's *playing the role* of father. One could find all of the fathers in a knowledge base simply by finding the set of men and then discarding those not known to fill the father role for some individual. The concept of a father clearly has its meaning compositionally dependent on the meaning of the `father` role.

In the history of KL-ONE-style languages, proposals have been made for a type of object called a "qua-concept" [Freeman, 1982], which would be a concept whose meaning is dependent on some role. `FATHER` as a qua-concept would have a slightly different structure than, say, `MAN`, reflecting the dependence of someone's being a father on the existence of another individual (some interesting property inheritance can be done in this case as well). CLASSIC, however, has no facility for this, so the best one can do is adhere to some reasonable conventions. If a separate concept for the role `father` is truly necessary (e.g., to act as a value restriction for some other role), consider naming it `MAN-qua-father`, to indicate the functional dependence. This concept would be a subconcept of `MAN`, and it could be made to work as if it were a qua-concept through the use of a procedural test, so that at least classification of all fathers could be achieved automatically.[14]

In the case of a `WINE`'s `grape`, one could use the same name for the role and the concept without resorting to any other mechanism. CLASSIC will not get confused; however, users

[14] What will be missing in this case is the automatic recognition that an `OLD-FATHER` is a `FATHER`, since no subsumption is computed on test functions (assuming `FATHER` and `OLD-FATHER` each had a single test function to compute their membership).

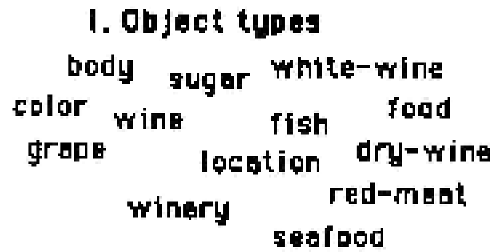
might. Thus, for clarity, it might be safer either to preface the role name with "has" to clearly distinguish the two senses (i.e., `has-grape` would be a role of `WINE`), or to create a compound concept name so that the role name will be simple. In our sample knowledge base in Section 5.3, we do the latter, creating the category of a `WINE-GRAPE`, and using `grape` as a role for `WINE`. In many cases, there is a natural role name to use so that this problem will not even arise. Such is the case with a term like "vintage," where the value restriction of the `vintage` role for `WINE` would be `YEAR`. It is also not required in any way that the names of roles should be nouns. `made-from` would be a perfectly reasonable name for the role we have been calling `grape`.

Finally, one should in general consider using roles to represent *parts* of objects, *intrinsic properties* (e.g., the color of a wine), and *extrinsic properties* (e.g., the price of a wine, which is not an intrinsic feature, but rather set in some external way), as well as for functionally-defined terms like "vintage."

5.2 A Simple Knowledge Engineering Methodology for CLASSIC

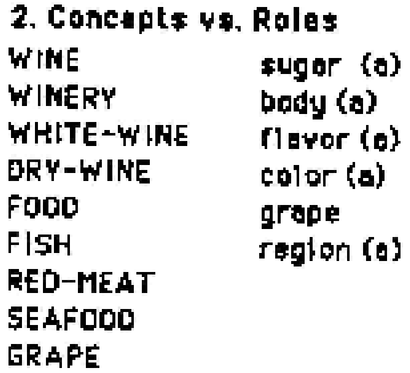
When attempting to analyze a domain and build a CLASSIC-style representation, it is often difficult to know how to begin. Over the years, we have developed some guidelines for building knowledge bases that break the process down into a series of steps, starting with a rough cut at the domain ontology and then refining the representation in several passes. While this method may oversimplify the knowledge representation process, it may be useful in many application areas, especially for those who are just getting started in using CLASSIC or other languages like it. We continue using our wine and meal examples. We have included below sketches of portions of the evolving KB to exemplify most of the steps.

1. **Enumerate Object Types.** First, without making any fine-grained distinctions, it is useful to try to write down a list of all types of objects you would ever care to make statements about or explain to a user. For example, important wine-related object-types will include *wine*; *grape*; *winery*; *location*; a wine's *color*, *body*, *flavor*, and *sugar-content*; different types of *food*, like *shellfish* and *red-meat*; subtypes of wine such as *white wine*; etc. The key thing initially is to get a comprehensive list of names without worrying about overlap between concepts or any properties that the items might have.

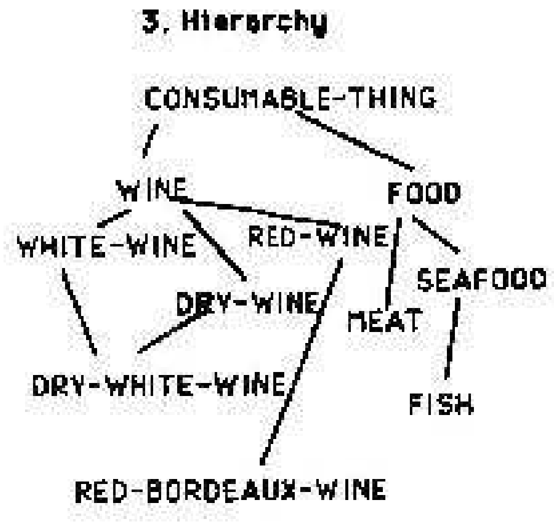


2. **Distinguish Concepts from Roles.** Looking at the list, make a major cut by distinguishing between objects that have independent existence and those that depend on other objects for their existence (see Section 5.1.2). The former will be concepts, the latter must be roles. For example, wines will exist as independent objects, as

will wineries, but the body of a wine and its sugar content are more appropriately thought of as roles. In developing a CLASSIC KB, it is also necessary to distinguish which roles are *attributes*, i.e., which ones have exactly one filler. Thus, `color` might be an attribute, since a given wine can have only one color, and `grape` would be a regular, multiply-fillable role, since a wine can be made from more than one type of grape.

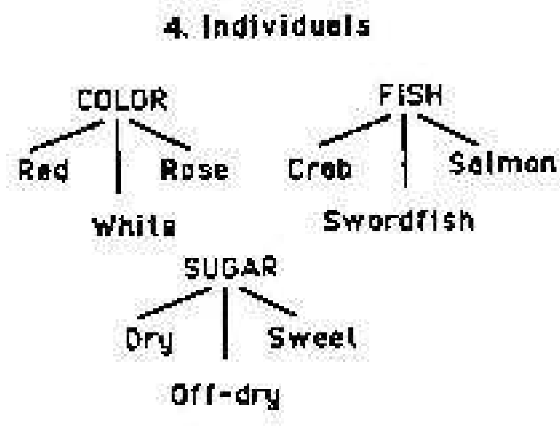


3. **Develop Concept Taxonomy.** Group the concept objects into a hierarchical taxonomy by asking if by being an instance of a type, an object will necessarily (i.e., by definition) be an instance of some other type. The latter will then be above the former in the hierarchy. For example, if something is a `WHITE-WINE`, it will necessarily be a `WINE`. Thus `WHITE-WINE` will be a descendant of `WINE` in the taxonomy. Remember that it is possible for a type to be an immediate descendant of more than one other type. For example, a `DRY-WHITE-WINE` must be both a `DRY-WINE` and a `WHITE-WINE`.[15]



[15]Note that once the final representation of a concept like `DRY-WHITE-WINE` is completed, CLASSIC will be able to determine automatically that it is a subconcept of the other two concepts. However, when developing the domain ontology, it is not a bad idea to sketch out these relationships by hand; once the formal representation is constructed and everything is classified, the user can check the resulting taxonomy against his/her original conception of the domain, to see if the formal representation is correct.

4. **Individuals.** Isolate the set of key individuals that will be important in all uses of the application. For example, wine colors like red, white, and rose, and wine sugar-levels like dry and sweet will be critical in the definition of concepts like `WHITE-WINE` and `DRY-WINE`. For each individual, try to determine all of the concepts that aptly describe it.



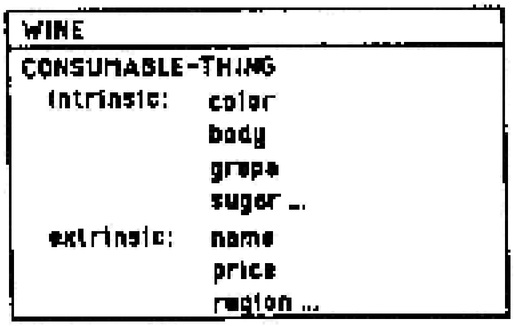
5. **Determine Properties and Parts.** Once the basic ontology is laid out, with the taxonomic relationships between concepts being fairly clear, it is time to turn attention to the internal structure of the concepts. For each concept enumerated so far, write down a list of its properties. These should include

   - *"intrinsic" properties* like the color and body of a wine;
   - *"extrinsic" properties* like a wine's name and its price;
   - *parts*, if the type of object is structured; these can be both physical and abstract "parts" (e.g., the courses of a meal, the grape of a wine, the casks of a winery). (In the case of wines, we have no intuitively obvious parts.)

   Record also any key relationships between individual members of the class and other items (e.g., relationships like employee that might not be considered properties or parts of a winery). Each of the above relationships should be assigned to a role (while it is useful to distinguish between parts and other properties, CLASSIC and related languages do not have any formal mechanism for distinguishing amongst different types of roles). It is reasonable to expect that many of the roles will be used in many concepts. Each of the items determined to be a role in Step 2 should be accounted for.

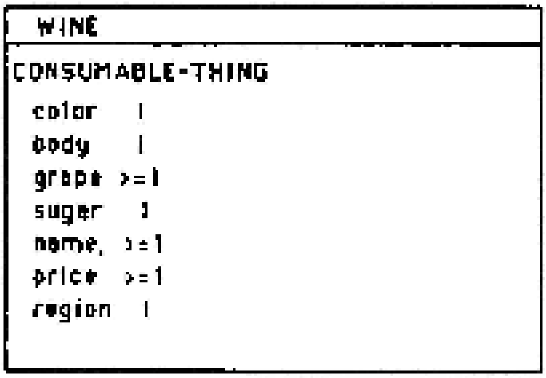
   N.B. Some of the roles determined to be relevant to a concept in this step will ultimately end up playing a part in the *definition* of the concept, and some will be used to express *derived* properties. In other words, some of the role restrictions generated in Steps 6–9 must be satisfied for an individual to be considered to satisfy a concept; the other restrictions will be appropriate to infer about the individual once it is determined to satisfy the concept definition. For example, a value restriction like **(FILLS** `color White)`, derived in Step 7, will be part of the meaning of `WHITE-WINE`; this means that an individual will need to have its color be provably white before it will be placed in that category. The same restriction could, however, be a derived property of CHARDONNAY-WINE, since it is not necessary to determine that a wine is white before deciding that it is a Chardonnay (it need only be known that it is made with a Chardonnay grape). Keep this in mind for Step 10. Also, see Section 4.2 for more on this distinction.

5. Properties



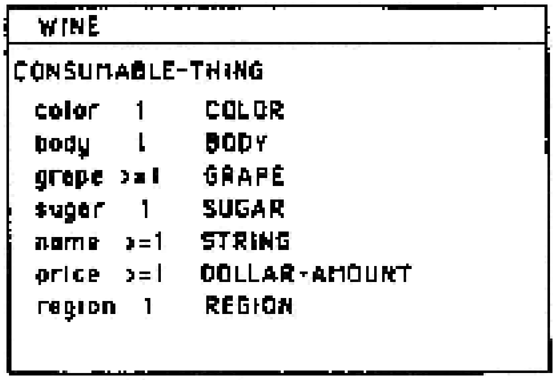
6. **Determine Number Restrictions.** For each concept and each role that is relevant to its meaning, determine the cardinality of the set of role fillers (e.g., that a wine can have only one region but several grapes). These will be expressed in CLASSIC as **AT-LEAST** and **AT-MOST** restrictions.

6. Cardinality



7. **Determine Value Restrictions.** For each concept and each of its relevant roles, determine the class of values that can appropriately fill the role. These "value restrictions" (e.g., that the `region` of a `WINE` must be a geographic region) will be expressed in **ALL** restrictions. In the event that a role must be filled by a single individual (e.g., a `CHARDONNAY-WINE` must have its `grape` role filled by exactly `Chardonnay`), or a fixed set of individuals, use the **FILLS** construct in conjunction with an **AT-MOST** restriction. If there is more than one potential filler (not all of which must necessarily fill the role), but the set of candidates is a fixed set of individuals, use the **ONE-OF** construct (e.g., a `NON-SWEET-WINE` has as the fillers of its `sugar` role the set (**ONE-OF** `Dry Off-Dry`)).

7. Value Restrictions

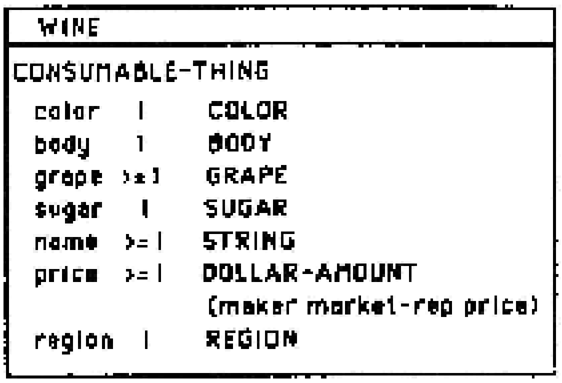


7. More Value Restrictions

8. **Detail Unrepresented Value Restrictions.** For each value restriction thus needed, make sure that the appropriate concept exists in the previously-generated general taxonomy. If it had previously been proposed, add it to the general taxonomy (for example, it is probable that we had not thought to create the concept of a geographic region prior to thinking about the structure of `WINE`). If the concept will be important in the domain model, go through all of the above steps for that new concept and any related ones you neglected to create before. For example, if you determine that the `grape` of a `WINE` must be a `WINE-GRAPE`, and the concept of such a grape is important, consider creating specialized subconcepts that might be useful (e.g., `CHARDONNAY`). For each of the new concepts, consider their properties and relations to other concepts and individuals.

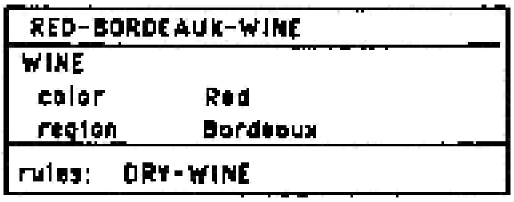
9. **Determine Inter-role Relationships.** For each concept, enumerate any relationships among its roles that might be important to your domain knowledge (for example, it might be important to restrict the `suggested-retail-price` of a `WINE` such that it is the `WINE`'s `maker`'s `marketing-rep` that sets it). CLASSIC and languages like it have only limited means of expressing these inter-role restrictions, but they are useful to enumerate. For a CLASSIC representation, any constraint that can be expressed as an equality between two chains of attributes on the same object can be expressed with the **SAME-AS** construct. Any other constraints must be expressed in opaque form with the **TEST-C** or **TEST-H** construct.

9. Constraints



10. **Distinguish Essential and Incidental Properties.** At this point, for each concept, we will have determined a set of parent concepts (expressed in the taxonomy) and a set of restrictions, namely number, value, inter-role equality (**SAME-AS**), and opaque test restrictions. For each concept, look over this set, think about what it would mean to be a member of the class specified by the concept, and isolate the set of concepts and restrictions that would appropriately constitute a set of essential properties. These properties would be sufficient for determining membership in the concept in question. For example, with a `RED-BORDEAUX-WINE`, the fact that it is a `WINE` whose `color` is `Red` and whose `region` is `Bordeaux` would be essential to its definition. It `sugar` content would be an incidental property and would not be necessary to know before determining that something was a red Bordeaux. The essential properties would constitute the definition of the concept while the other properties would then be expressed as the consequents of *rules* associated with the concept (e.g., `RED-BORDEAUX-WINE` would have a rule asserting that the wine is a `DRY-WINE`; thus the sugar content need not be known in order to determine that something is a red Bordeaux, but it would be universally true of all red Bordeaux wines).

10. Rules



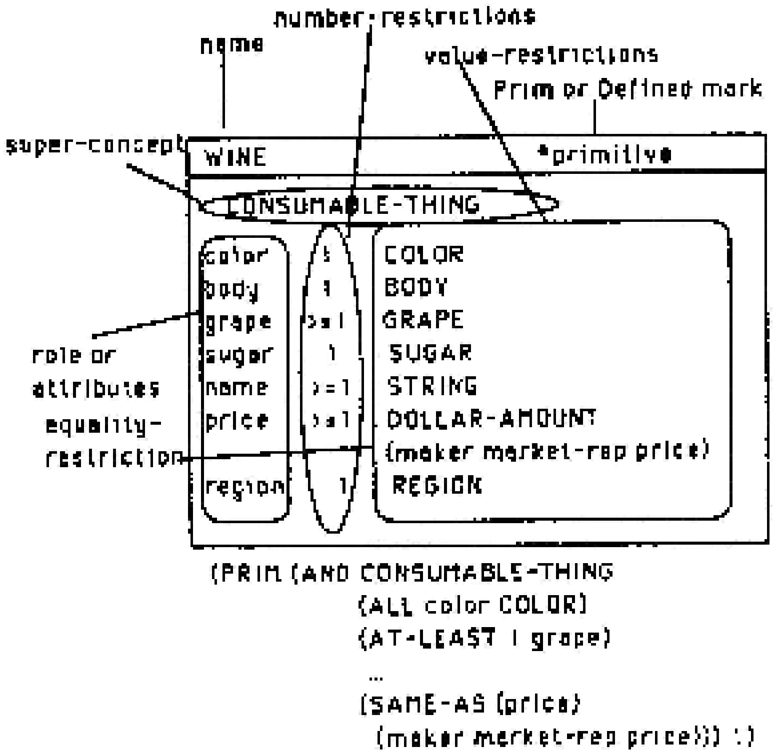
11. **Distinguish Primitive and Defined Concepts.** Determine if each proposed concept definition is complete. That is, do the conditions determined by the above steps constitute a complete set of necessary and sufficient conditions for the concept? In the case of a `RED-BORDEAUX-WINE`, the conditions that it is a `WINE`, that its `color` must be exactly `Red`, and that its `region` must be `Bordeaux` would indeed be both necessary and sufficient. For those items whose definitional complement is not fully sufficient, make the concepts primitive. For example, we may not consider `WINE` to be fully defined as a `POTABLE-LIQUID` with at least one `grape`; we would not want every liquid made from grapes to be considered a wine. Thus `WINE` would have to be

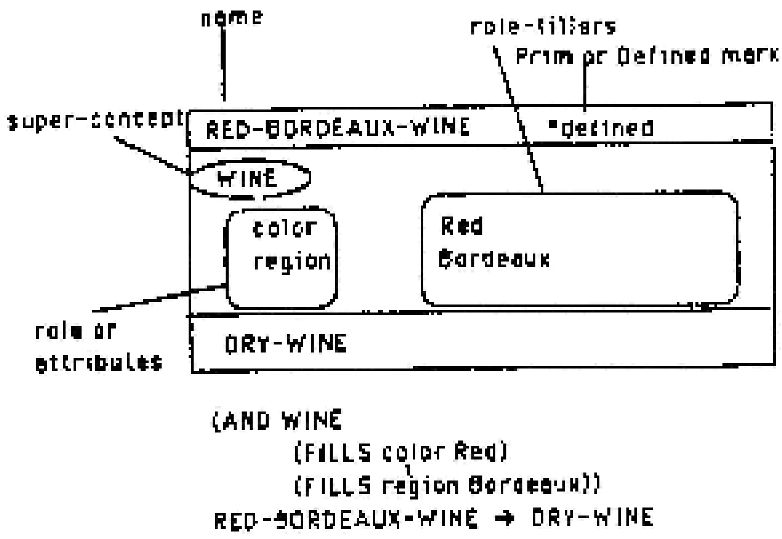
primitive.

12. **Determine DISJOINT-PRIMITIVE Concepts.** For those concepts determined to be primitive, determine if any are mutually exclusive. Group those so determined into clusters under a common superconcept.[16] Typically, the highest concepts in the hierarchy will be primitive and disjoint. For example, `SHELLFISH` and `FISH` would be good candidates for disjoint primitive concepts with a mutual parent of `SEAFOOD`. They are disjoint because no individual can be described by more than one of them at a time, and they are primitive because in this domain we are not interested in any internal structure or further description of individuals that satisfy these descriptions (we will typically declare by fiat that `Crab` is a `SHELLFISH`, without expecting CLASSIC to be able to determine it by itself). Use the **DISJOINT-PRIMITIVE** construct to specify these concepts.

The result of translating the informal representation created above into CLASSIC will be a knowledge base of concepts, roles, individuals, and rules (note that an item must be defined prior to its first use, since there are no circular definitions allowed; however, a concept can be used in the consequent of a rule that is associated with it). The concepts will have a set of necessary, and sometimes, necessary and sufficient conditions expressed as sets of more general concepts and restrictions (those concepts with no sufficient conditions would be constructed using the **PRIMITIVE** or **DISJOINT-PRIMITIVE** operators). The parents and restrictions on a concept would be conjoined with the **AND** operator, and each restriction would be expressed with an **ALL**, **AT-LEAST**, **AT-MOST**, **FILLS**, or **SAME-AS** operator (or a **TEST**, if appropriate). Named concepts would also be the antecedents of rules expressing necessary conditions—descriptions that would follow once something were determined to be a member of the class. Here we show both the schematic form and the CLASSIC form of the two examples we have been following:

[16] In CLASSIC, there can be several disjoint groupings under the same concept, with the assumption that there is a common dimension along which all the items in a grouping differ (imagine, for example, grouping subconcepts of `PERSON` by gender or by age). Thus the **DISJOINT-PRIMITIVE** construct requires the user not only to specify the parent concept, but to name a grouping into which to put the primitive being specified.





5.3 A Sample Knowledge Base

In order to illustrate the general ways a user will use CLASSIC to build a knowledge base, we will now consider some sample definitions from the world of wines and meals. The basic goal here is to allow a user to describe the food eaten at a particular course of a meal (in a very simple way), and have the KB recommend an appropriate set of wines. The knowledge is organized so that a new wine can be described in a number of different ways (e.g., it might be asserted to be a late-harvest Semillon, or a white wine from the Loire region); it is then classified with respect to a general set of useful wine-types (e.g., CHARDONNAY-WINE). Once the wine is classified, properties not directly asserted by the user are derived using rules whose antecedents are the general wine-types (e.g., if all Chardonnays are either full- or medium-bodied, this information will be represented as a rule whose antecedent is the concept of a Chardonnay wine). Thus wines can be entered in a variety of ways—by region, by varietal, by color, etc.—and ultimately as much as possible about their color, body, sweetness, etc., will be ascertained automatically.

In parallel to the hierarchy of useful wine-types, we have a simple hierarchy of food-types. The food-types are used to describe a course the user is considering having (e.g., "a MEAL-COURSE[17] whose food is a SHELLFISH"). The connection between wines and food is to be made via a hierarchy of course-types (e.g., SHELLFISH-COURSE). Each useful course-type (not every possible course-type forces a choice of wines) has an associated rule that states what characteristics are required of its wine (e.g., seafood-courses demand white wines, oyster-courses need sweet wines). The system makes a "recommendation" in a simple forward-chaining way: the user's course-type is classified, rules applying to it are inherited from all descriptions that apply to the course, the rules are fired, and the consequents assert various constraints on the drink of the course. The user can then examine the drink role of the course to see what characteristics are necessary for the wine, as well as which wines are compatible with those characteristics. This is an example of a simple forward-chaining constraint propagation application. The value of organizing the knowledge in this fashion is that the wine descriptions are decoupled from the requirements for each course type. A new wine can be added, a given wine can easily have its characteristics changed, or a given food can be associated with different wine characteristics, all by making only local changes.

Figure 2 shows the top few levels of the concept hierarchy for our wine and food KB.

In the subsequent figures illustrating our concepts, we do not present the information in the exact form in which we would type it to CLASSIC—that would involve for each item a call to a COMMON LISP function. Instead, we have used the notation in Figure 3 to signify the type of description being defined or applied.

Thus, for example,

WINE-COLOR $\Leftrightarrow$ (**AND** WINE-PROPERTY (**ONE-OF** White Rose Red))

would mean that WINE-COLOR is fully defined as a WINE-PROPERTY whose only possible instances are White, Rose, and Red. Similarly,

SEAFOOD-COURSE $\supset$ (**ALL** drink WHITE-WINE)

would mean that if an object were determined to be a SEAFOOD-COURSE, it automatically follows that all of its drinks are WHITE-WINEs. The two definitions,

MEAL-COURSE $\otimes_1^{+}\Rightarrow$ CONSUMABLE-THING,
(**AND** (**AT-LEAST** 1 food)
(**ALL** food EDIBLE-THING)
(**AT-LEAST** 1 drink)
(**ALL** drink POTABLE-LIQUID))

[17]In the sample KB, we use course for the role of a course at a meal, and MEAL-COURSE for the concept of a course.

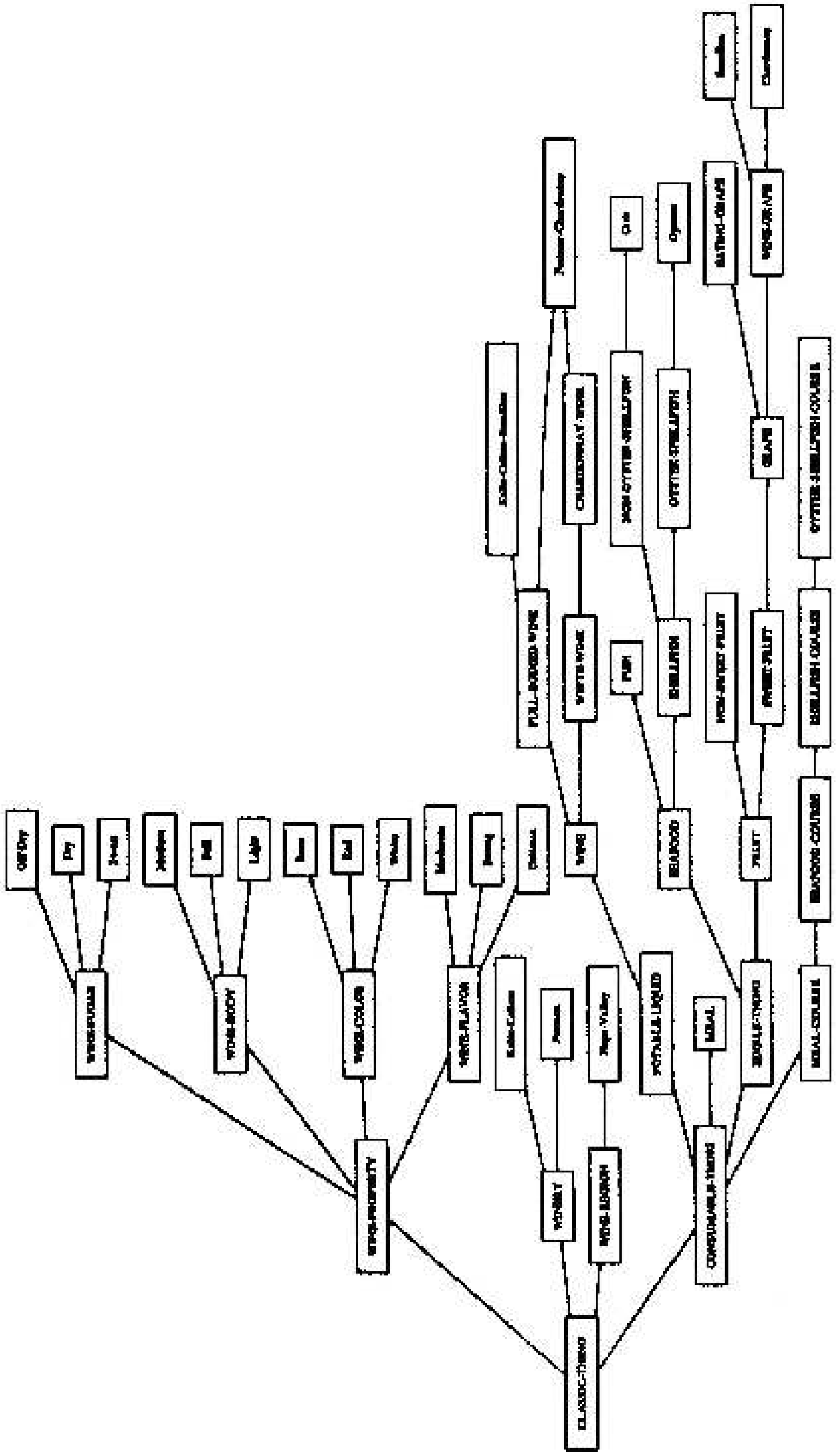


Figure 2: Hierarchy of the Sample Knowledge Base

expression	*meaning*
c $\Leftrightarrow$ e	c is fully defined by the expression e
c $\Rightarrow$ e	c is a primitive subconcept of the concept represented by e
c $\otimes_i\Rightarrow$ e	c is a disjoint primitive subconcept of the concept represented by e, in the disjoint grouping labeled "i"
c $\otimes_i^+\Rightarrow$ e, E	c is the combination of the expression E and a (unnamed) disjoint primitive subconcept of the concept represented by e (which is in the disjoint grouping labeled "i")
c $\supset$ e	c is the antecedent of a rule whose consequent is e
i $\rightarrow$ e	i is an individual and is asserted to have the properties described by e
r $\mapsto$	r is a role
r $\overset{!}{\mapsto}$	r is an attribute

Figure 3: Symbols Used to Describe the Sample Knowledge Base

and

```
MEAL   ⊗₁⁺⇒   CONSUMABLE-THING,
              (AND (AT-LEAST 1 course)
                   (ALL course MEAL-COURSE))
```

would mean that MEAL-COURSE and MEAL were both specializations of CONSUMABLE-THING, they were mutually disjoint, and they each had the additional properties specified.

Figure 4 illustrates the beginning of our wine and food KB. Since roles are used in concept definitions, and, in the current version of CLASSIC, do not themselves depend on any other constructs, the roles to be used in the KB would be defined first. In this case, we assume the following roles are defined at the beginning of the KB: `color`, `body`, `flavor`, `sugar`, `region`, `grape`, `maker`, `drink`, `food`, and `course` (note that all but `grape` and `course` are attributes). After the roles are defined, it is usually a good idea to define the classes of objects that are used only in value restrictions of other concepts. In the figure, we define a simple primitive, WINE-PROPERTY, which will serve as the parent for all wine-properties later used in the KB.[18] Since we want wines to have colors, and we can specify in advance all of the individuals that can be wine colors, we create a defined subconcept of WINE-PROPERTY called WINE-COLOR, specifying all of its possible instances with a **ONE-OF** description. Similarly, we create the wine-properties of WINE-BODY, WINE-FLAVOR, and WINE-SUGAR.

Next we create the top part of the main hierarchy. Because concepts and roles must be defined before they are used, a CLASSIC KB file will generally proceed from most general concepts to most specific ones. In Figure 5, we define a few high-level primitive concepts. The simple world we are describing is broken into four disjoint parts: WINE-PROPERTYs, WINERYs, WINE-REGIONs, and CONSUMABLE-THINGs (this will be used for foods and wines,

[18] As illustrated, WINE-PROPERTY is a member of disjoint grouping number 1 of CLASSIC-THING. In Figure 5, we illustrate the other concepts that are disjoint from this one.

Wine and Meal Knowledge Base.

After defining the roles, define value restriction concepts and individuals to be used in further definitions.

`color` $\overset{!}{\mapsto}$
`body` $\overset{!}{\mapsto}$
`flavor` $\overset{!}{\mapsto}$
`sugar` $\overset{!}{\mapsto}$
`region` $\overset{!}{\mapsto}$
`grape` $\mapsto$
`maker` $\overset{!}{\mapsto}$
`drink` $\overset{!}{\mapsto}$
`food` $\overset{!}{\mapsto}$
`course` $\mapsto$

`WINE-PROPERTY` $\otimes_1\Rightarrow$ `CLASSIC-THING`

`WINE-COLOR` $\Leftrightarrow$ (**AND** `WINE-PROPERTY` (**ONE-OF** `White Rose Red`))

`WINE-BODY` $\Leftrightarrow$ (**AND** `WINE-PROPERTY` (**ONE-OF** `Light Medium Full`))

`WINE-FLAVOR` $\Leftrightarrow$ (**AND** `WINE-PROPERTY` (**ONE-OF** `Delicate Moderate Strong`))

`WINE-SUGAR` $\Leftrightarrow$ (**AND** `WINE-PROPERTY` (**ONE-OF** `Sweet Off-Dry Dry`))

Figure 4: Sample Knowledge Base — Roles and Some Basic Value Restrictions

Define the other topmost concepts.

$\texttt{WINERY} \quad \otimes_1\!\Rightarrow \quad \texttt{CLASSIC-THING}$
$\texttt{WINE-REGION} \quad \otimes_1\!\Rightarrow \quad \texttt{CLASSIC-THING}$
$\texttt{CONSUMABLE-THING} \quad \otimes_1\!\Rightarrow \quad \texttt{CLASSIC-THING}$

$\texttt{EDIBLE-THING} \quad \otimes_1\!\Rightarrow \quad \texttt{CONSUMABLE-THING}$
$\texttt{POTABLE-LIQUID} \quad \otimes_1\!\Rightarrow \quad \texttt{CONSUMABLE-THING}$

$\texttt{SEAFOOD} \quad \otimes_1\!\Rightarrow \quad \texttt{EDIBLE-THING}$
$\texttt{FRUIT} \quad \otimes_1\!\Rightarrow \quad \texttt{EDIBLE-THING}$

$\texttt{SHELLFISH} \quad \otimes_1\!\Rightarrow \quad \texttt{SEAFOOD}$
$\texttt{FISH} \quad \otimes_1\!\Rightarrow \quad \texttt{SEAFOOD}$

Define some instances of `WINERY` and `WINE-REGION`.

$\texttt{Forman} \quad \rightarrow \quad \texttt{WINERY}$
$\texttt{Kalin-Cellars} \quad \rightarrow \quad \texttt{WINERY}$
$\texttt{Napa-Valley} \quad \rightarrow \quad \texttt{WINE-REGION}$

Define `WINE-GRAPE` and some instances of it.

$\texttt{SWEET-FRUIT} \quad \otimes_1\!\Rightarrow \quad \texttt{FRUIT}$
$\texttt{GRAPE} \quad \otimes_1\!\Rightarrow \quad \texttt{SWEET-FRUIT}$
$\texttt{EATING-GRAPE} \quad \Rightarrow \quad \texttt{GRAPE}$
$\texttt{WINE-GRAPE} \quad \Rightarrow \quad \texttt{GRAPE}$
$\texttt{Chardonnay} \quad \rightarrow \quad \texttt{WINE-GRAPE}$
$\texttt{Semillon} \quad \rightarrow \quad \texttt{WINE-GRAPE}$

Figure 5: Sample Knowledge Base — More General Concepts

as well as meals and courses—special categories needed to trigger the inferences about wine-types for different foods—see Figure 9). In this figure we also define several types of `CONSUMABLE-THING`, and then some representative instances of `WINERY` and `WINE-REGION`. We then include some information about grapes that will be needed later. Note that there may be some grapes used for eating that are also used for making wine, so `EATING-GRAPE` and `WINE-GRAPE` have been defined as primitive but not disjoint concepts under `GRAPE`.

So far, we have created only simple primitive concepts. CLASSIC allows the construction of much more complex, but still primitive concepts. For example, we might want to give `WINE` some complex necessary conditions as part of its meaning, but tell CLASSIC that the conditions we give it are not sufficient for recognizing wines. We would accomplish this by defining `WINE` as a primitive with a complex expression, as illustrated in Figure 6. We can read this definition of `WINE` as something like, "a wine is, among other things, a potable liquid with exactly one color [because `color` is an attribute], which must be a wine-color, exactly one body, ..."

Once we have the key basic concept of a `WINE` defined, we can describe the more specialized types of wines we would like to be able to recognize automatically. Figure 7 illustrates three fully-defined wine subconcepts. For example, a `WHITE-WINE` is fully defined as a wine whose color is white. The condition that the color of a `WHITE-WINE` must be exactly

Define the concept of a wine.

```
WINE  ⇒  (AND POTABLE-LIQUID
              (AT-LEAST 1 color)
              (ALL color WINE-COLOR)
              (AT-LEAST 1 body)
              (ALL body WINE-BODY)
              (AT-LEAST 1 flavor)
              (ALL flavor WINE-FLAVOR)
              (AT-LEAST 1 sugar)
              (ALL sugar WINE-SUGAR)
              (AT-LEAST 1 region)
              (ALL region WINE-REGION)
              (AT-LEAST 1 grape)
              (ALL grape WINE-GRAPE)
              (AT-LEAST 1 maker)
              (ALL maker WINERY))
```

Figure 6: Sample Knowledge Base — WINE

White is sufficiently stated as (**FILLS** color White), since color has been defined as an attribute, and an attribute has exactly one filler. In the case of CHARDONNAY-WINE, whose grape role must be filled by exactly the individual Chardonnay, the definition of WINE says that a CHARDONNAY-WINE (or any wine) must have at least 1 grape, and the restriction on CHARDONNAY-WINE, (**ALL** grape (**ONE-OF** Chardonnay)), specifies what that grape is, and that there can be no additional fillers for the grape role. Returning to WHITE-WINE, note that White is consonant with the general value restriction previously stated for the color role of WINE (i.e., White is a WINE-COLOR). In any case, CLASSIC will recognize any wine whose color is determined to be white *by any means* (user assertion, rule firing, propagation from some other assertion, etc.) as an instance of WHITE-WINE.

Figure 7 also illustrates some rules based on CHARDONNAY-WINE. Since we have stated nothing specific about the Chardonnay grape (i.e., it is never stated that wines made from this grape are white), we have a rule stating that Chardonnays have color white. Thus, any wine whose grape is recognized to be exactly Chardonnay will end up being a WHITE-WINE as well, since the rule will assert that its color is white, and classification will use the fact that it is a wine and also white to determine that it is a WHITE-WINE.[19] We also include rules about the body and flavor of Chardonnays.

Once we have the wine hierarchy defined, it is reasonable to create and describe individuals for various particular wines. Figure 8 illustrates two typical descriptions of such individuals. Note that by inheritance Forman-Chardonnay will end up with all known properties of CHARDONNAY-WINEs (as well as of WINEs in general), as well as the individual properties stated in the figure. As we mentioned above, once this individual is created, it will be classified under all appropriate defined concepts, such as FULL-BODIED-WINE. In addition, in this case, the rule that says that Chardonnays are always white will fire, and

[19]Note that if we had included the white color restriction as part of the definition of CHARDONNAY-WINE, it would have made that restriction one of the conditions necessary for a wine to have before it could be determined to be a Chardonnay wine. Since the essential property of being a Chardonnay wine is having the right grape, then the white color is a derivative property that should not be included in the basic concept definition. Thus we use a rule to assert that Chardonnays are white.

Define some subcategories of wines.

A white wine is a wine whose color is white.
`WHITE-WINE ⇔ (AND WINE (FILLS color White))`

A full-bodied wine is a wine whose body is full.
`FULL-BODIED-WINE ⇔ (AND WINE (FILLS body Full))`

A `CHARDONNAY-WINE` is a wine with exactly one grape, which is `Chardonnay`.
`CHARDONNAY-WINE ⇔ (AND WINE (ALL grape (ONE-OF Chardonnay)))`

Now assert some rules about Chardonnay wines.

Chardonnays are always white.
`CHARDONNAY-WINE ⊃ (FILLS color White)`

Chardonnays are always either full- or medium-bodied wines.
`CHARDONNAY-WINE ⊃ (ALL body (ONE-OF Full Medium))`

Chardonnays are not delicate.
`CHARDONNAY-WINE ⊃ (ALL flavor (ONE-OF Strong Moderate))`

Figure 7: Sample Knowledge Base — Defined Wine Subconcepts

`Forman-Chardonnay` will end up being classified as a `WHITE-WINE` as well. Also note that `Kalin-Cellars-Semillon` is only partially described, in that we have stated that one of its `grapes` is `Semillon`, but have not closed the `grape` role.

In Figure 9 we illustrate some simple primitive concepts that will appear below `SHELLFISH` in the hierarchy. We then represent the concepts `MEAL-COURSE` and `MEAL` as disjoint primitive concepts under `CONSUMABLE-THING` (disjoint also from `POTABLE-LIQUID` and `EDIBLE-THING`—see Figure 5), but having complex structure. A `MEAL-COURSE` is defined as having exactly one food and exactly one drink (recall that `food` and `drink` are attributes), while a `MEAL` is defined as having at least one course. In this simple application, the type of food served at a course will be stated directly; the categorization of the course on the basis of this food will then be used to trigger rules constraining the properties of any wine served. Thus, the food concepts and individuals need no internal structure. A course individual will be classified under a specific type of course (e.g., `SEAFOOD-COURSE` in Figure 10) as soon as its food is known, and the drink role will be used to accumulate properties of the wine for the given course.

Finally, to allow our knowledge base to perform the appropriate inferences when we describe an individual course, we will need a set of rules that constrain the type of wine to be drunk with each appropriate food-type. In some cases, we can have very general rules, such as seafood requiring white wines, and in others we can have very narrowly applicable ones, such as oysters requiring sweet wines. Each rule is associated with the appropriate concept in the KB, as illustrated in Figure 10. When a given course is described (such as `Course-256` in the figure), all rules that apply will be inherited and triggered. In the case of `Course-256`, since the food of the course is oysters, the course will be classified as an `OYSTER-SHELLFISH-COURSE`: because `food` is an attribute, the `food` role of `Course-256` is closed as soon as it is asserted that it is filled with `Oysters`; with `Oysters` as the only

Create and describe some individual wines.

```
Forman-Chardonnay  →  (AND CHARDONNAY-WINE
                           (FILLS body Full)
                           (FILLS flavor Moderate)
                           (FILLS sugar Dry)
                           (FILLS maker Forman))

Kalin-Cellars-Semillon  →  (AND WINE
                                (FILLS grape Semillon)
                                (FILLS body Full)
                                (FILLS flavor Strong)
                                (FILLS sugar Dry)
                                (FILLS maker Kalin-Cellars))
```

Figure 8: Sample Knowledge Base — Individual Wines

Define some primitive food-types.

```
OYSTER-SHELLFISH  ⊗₁⇒  SHELLFISH
NON-OYSTER-SHELLFISH  ⊗₁⇒  SHELLFISH
```

Create some instances of foods.

```
Oysters  →  OYSTER-SHELLFISH
Crab  →  NON-OYSTER-SHELLFISH
```

Define the concepts for a course and a meal

```
MEAL-COURSE  ⊗₁⁺⇒  CONSUMABLE-THING,
                   (AND (AT-LEAST 1 food)
                        (ALL food EDIBLE-THING)
                        (AT-LEAST 1 drink)
                        (ALL drink POTABLE-LIQUID))

MEAL  ⊗₁⁺⇒  CONSUMABLE-THING,
            (AND (AT-LEAST 1 course)
                 (ALL course MEAL-COURSE))
```

Figure 9: Sample Knowledge Base—Foods, Meals, and Courses

Define some concepts that allow recognition of course-types, which will be antecedents of rules constraining wines.

SEAFOOD-COURSE $\Leftrightarrow$ (**AND** MEAL-COURSE (**ALL** food SEAFOOD))

Note that SHELLFISH-COURSE will be classified under SEAFOOD-COURSE and OYSTER-SHELLFISH-COURSE will be classified under SHELLFISH-COURSE:

SHELLFISH-COURSE $\Leftrightarrow$ (**AND** MEAL-COURSE (**ALL** food SHELLFISH))

OYSTER-SHELLFISH-COURSE $\Leftrightarrow$ (**AND** MEAL-COURSE (**ALL** food OYSTER-SHELLFISH))

Now assert rules pertaining to course-types:

SEAFOOD-COURSE $\supset$ (**ALL** drink WHITE-WINE)

```
SHELLFISH-COURSE  ⊃  (ALL drink
                           (AND (FILLS body Full)
                                (ALL flavor (ONE-OF Moderate Strong))))
```

OYSTER-SHELLFISH-COURSE $\supset$ (**ALL** drink (**FILLS** sugar Sweet))

Create a specific course with oysters as food.

Course-256 $\rightarrow$ (**AND** MEAL-COURSE (**FILLS** food Oysters))

Figure 10: Sample Knowledge Base — Course-types with Rules

filler of that role, the **ALL** restriction on OYSTER-SHELLFISH-COURSE is satisfied. The classification of Course-256 as an OYSTER-SHELLFISH-COURSE makes applicable all rules from OYSTER-SHELLFISH-COURSE, SHELLFISH-COURSE, and SEAFOOD-COURSE. The drink of Course-256 will thus be constrained to be a sweet (from OYSTER-SHELLFISH-COURSE), full-bodied (from SHELLFISH-COURSE) white wine (from SEAFOOD-COURSE), of either moderate or strong flavor (from SHELLFISH-COURSE).

6 Tricks of the Trade

The expressive limitations of CLASSIC mean that there are many things that it cannot directly represent. After building a number of knowledge bases using the system, we have found some ways of getting around some of these expressive limitations.

The reason these techniques are presented in a separate section is that the meanings that CLASSIC places on the resultant concepts are different than their intuitive meanings. Under some circumstances CLASSIC will act in a way inconsistent with the intuitive meanings. Often this divergence only shows up when certain types of extra information are added to the knowledge base—if this extra information is never added, then CLASSIC will adhere to the intuitive behavior. (For example, see the first way of representing a limited form of negation in Section 6.1.) Therefore, the knowledge base designer must be extremely careful when using these techniques.

These techniques are most useful when used sparingly. If a designer finds it necessary to use a large number of these "tricks," then perhaps CLASSIC should not be used for his/her application.

6.1 Negation and Complements

As noted before, there is no full negation in CLASSIC, but there are a few ways to represent limited forms of negation or complements.

One method can be used to define the concept of non-sweet wines. Given that wines have exactly one filler for their `sugar` role, and that the only possible fillers for the `sugar` role of wines are `Dry`, `Off-Dry`, and `Sweet`, a non-sweet wine can be defined as

(**AND** `WINE` (**ALL** `sugar` (**ONE-OF** `Dry Off-Dry`))).

Since `WINE-SUGAR` has exactly three instances, this concept is the complement of sweet wines (`WINE`s with filler `Sweet` for their `sugar` role) in the universe of wines.

However, this trick does not work as well when a restriction is based on a primitive concept (i.e., `WINE-GRAPE`) and not on a **ONE-OF** concept (i.e., `WINE-SUGAR`). A non-Chardonnay wine can be defined as

(**AND** `WINE` (**ALL** `grape` (**ONE-OF** `Semillon`))),

since `Chardonnay` and `Semillon` are the only grapes in the KB. However, if a new grape is added to the KB (i.e., `Riesling`), then this definition would no longer represent the wines made from all grapes except Chardonnay.

Another form of negation can be represented with disjoint primitives. If the concepts `FISH` and `SHELLFISH` are disjoint primitives under the concept `SEAFOOD`, then there can be no individuals belonging to both `FISH` and `SHELLFISH`. However, in this situation, it is possible for something to be a seafood and neither a fish nor a shellfish, and thus `FISH` is not exactly the relative complement of `SHELLFISH` with respect to `SEAFOOD`.

Finally, test concepts can also be used to capture part of the meaning of complements. A test function that returns false if an individual satisfies some concept, true if the individual cannot possibly satisfy it, and unknown otherwise, can be used to create a complement concept. However, there is a small problem with this method of complementation. The complement concept will not be recognized as disjoint from the other concept, so, for instance, the conjunction of the two concepts will not be considered incoherent, although it cannot, in reality, have any instances.

6.2 Disjunction

Although there is no "**OR**" operator in the CLASSIC language, disjunction can be captured in some special cases.

The first of these is simply a **ONE-OF** concept, which provides an extremely simple and uninteresting case of disjunction (of the individuals in the set). The second case builds on the first by using the **ONE-OF** concept in a value restriction. For example, the concept

`(`**AND** `WINE (`**ALL** `grape (`**ONE-OF** `Semillon Sauvignon)))`

represents the disjunction of wines made from semillon grapes and wines made from sauvignon grapes. Once a disjunctive concept like this is formed with a **ONE-OF** embedded in an **ALL**, such a concept can in turn be used in another **ALL** restriction, thus allowing arbitrarily deep nesting.

The above types of disjunction are not really tricks at all. They represent true disjunction—however, only certain, very limited, types of disjunction can be represented this way.

General disjunction can be crudely approximated, however, by using a simple trick. When one concept subsumes others, then it subsumes their disjunction, and can, under some circumstances, act like their disjunction. For example, in Figure 5, `SEAFOOD` subsumes the disjunction of `SHELLFISH` and `FISH`. If no individuals become instances of `SEAFOOD` without becoming instances of either `SHELLFISH` or `FISH` then `SEAFOOD` can be considered to be the disjunction of `SHELLFISH` and `FISH`. Because there may be instances of `SEAFOOD` that are neither `SHELLFISH` nor `FISH`, this is not true disjunction. (Learning that an individual is not an instance of `FISH` does not make it an instance of `SHELLFISH`.)

6.3 Defaults

CLASSIC enforces a strict inheritance hierarchy and does not provide a default operator. However, a limited form of defaults can be represented with the aid of rules and test functions.

For example, to make wines have default color red, use a test function (perhaps called **no-known-color**) that returns true if the number of currently known fillers of the `color` role is zero, and false otherwise[20] and use it in the concept `WINE-CAUSE-DEFAULT-RED`, defined as

`WINE-CAUSE-DEFAULT-RED`	$\Leftrightarrow$	`(`**AND** `WINE (`**TEST-C** **no-known-color**`))`
`WINE-CAUSE-DEFAULT-RED`	$\supset$	`(`**FILLS** `color Red).`

This will cause wines that are not given a color to become red wines because they will pass the test function, become instances of `WINE-CAUSE-DEFAULT-RED`, and be given `color Red` as a result of the firing of the rule above.

[20] This is different from knowing that there are no *possible* fillers for the `color` role, as CLASSIC can represent individuals, such as instances of `WINE`, for which there must be a filler for a role without knowing the actual filler.

WARNING: Small changes to the implementation of CLASSIC could cause this trick to fail as it uses a test function that violates the conditions placed on test functions. (Test functions in CLASSIC should be monotonic, i.e., adding information cannot cause the result of a test function to change from true to false, or vice versa.) *Use this trick with extreme caution.*

6.4 More Powerful Rules

Rules are an important part of CLASSIC, but are limited in that the antecedent of a rule can only be a named CLASSIC concept. However, using test restrictions in the antecedent of rules allows arbitrary pattern-matching to determine rule applicability. For example, we might want to extend the wine example to consider vintages and then conclude that if some wine is from a good vintage year then it is expensive. The definition of "good vintage" might be quite complicated, and not expressible in CLASSIC without using a test restriction.

This method does not cause any particular problems, aside from the general problem inherent in the use of (opaque) test functions, as long as the test conforms to the conditions placed on test functions. However, excessive use of test functions can cause performance degradation if the test concepts end up near the top of the concept hierarchy, where their tests will be run frequently.

6.5 Integrity Checking

Rules can also be used to provide a sort of integrity checking, by using test concepts as their conclusions. In this case, once an individual is found to satisfy the antecedent of the rule, it is made an instance of the test concept. Part of this process is to run the test function on the individual; if the individual is inconsistent with the test function then the individual is also removed from the antecedent concept. In this way complicated integrity constraints can be created for otherwise test-free concepts.

For example, we might want to check that late-harvest grapes have a sugar content of at least 30. This can be done by creating a rule

`LATE-HARVEST-GRAPE` $\supset$ **(TEST-C sugar-at-least-30)**

where **sugar-at-least-30** returns unknown if there is no currently known filler of the `sugar-content` role of a grape, true if the filler is known and is at least `30`, and false otherwise.

This is different from including the test condition as part of `LATE-HARVEST-GRAPE` in two ways. First, the test does not become part of the definition of the concept so it will not be subsumed by another concept that happens to incorporate the same test. Second, if `LATE-HARVEST-GRAPE` is a defined concept then individuals can be recognized as its instances without passing the test; they are forced to be (and remain) consistent with the test.

6.6 Restrictions on Roles

The CLASSIC language supports restrictions of the form "all of the drinks in a picnic basket are wines," and "a picnic basket has at least one drink," but there is no operator for saying precisely "at least two of the drinks in a picnic basket are white wines."[21]

When this sort of restriction is needed, a test can be used. For example, a test function to determine if at least two of the drinks in a picnic basket are white wines can be written as follows:

- if there are two known fillers of the `drink` role of the picnic basket that are instances of `WHITE-WINE` then return true;
- otherwise, if there can be at most one filler of the `drink` role of the picnic basket, then return false;
- otherwise, if all the fillers of the `drink` role of the picnic basket must be white wines because the type of its drinks is subsumed by `WHITE-WINE`, and there must be at least two drinks for the picnic basket, then return true;
- otherwise, if all the picnic basket's drinks are known, then if there is at most one of them that is an instance of `WHITE-WINE` then return false;
- otherwise return unknown.

As with all tests, CLASSIC treats the function as a black box, and will not discover any subsumption relationships between different test functions. This can pose a problem here because there are a large number of possible subsumption relationships between these sorts of restrictions. For example, "at least two of the drinks in a picnic basket are white wines" subsumes "at least three of the drinks in a picnic basket are full-bodied white wines," but CLASSIC cannot discover these relationships, which depend on the behavior of test functions.

Further, this test function contains a potentially dangerous "closed-world assumption" in that it assumes that a drink that is not known to be an instance of `WHITE-WINE` will never be an instance of `WHITE-WINE`. Since CLASSIC allows the acquisition of extra information about individuals, it is possible that a drink could later become an instance of `WHITE-WINE`, thus invalidating the conclusion drawn by this test function.

6.7 Dummy Individuals

As mentioned in Section 3.1, CLASSIC can answer queries about mandated properties of fillers of roles without knowing the identity of the fillers. Some of these queries can be answered by getting the value restriction for the role. For example, `Course-256` from Figure 10 must have the `body` of all its `drinks` be `Full`, since it has the property

(**ALL** `drink` (**FILLS** `body Full`)),

by virtue of its being a `SHELLFISH-COURSE`. This can be determined by CLASSIC without knowing the actual `drink` of the course.

However, this method does not pick up the rules that might be applied to the role filler. For example, under the definitions,

[21] This is a deliberate omission, as the inclusion of such operators makes determining subsumption computationally intractable [Nebel, 1988].

```
KOSHER-WEDDING  ⇔  (AND WEDDING (ALL meal KOSHER-MEAL))
KOSHER-MEAL     ⊃  (ALL course (ALL drink KOSHER-WINE))
Lori's-Wedding  →  KOSHER-WEDDING,
```

the value restriction for the `drink` of any `course` of the meal at `Lori's-Wedding` would not be known to be a `KOSHER-WINE`, even though the `meal` at `Lori's-Wedding` must be a `KOSHER-MEAL` and there is a rule on `KOSHER-MEAL` asserting that all the `drink`s of each `course` must be `KOSHER-WINE`s. To pick up this restriction it is necessary to create a "dummy" meal for `Lori's-Wedding`. Then the rule will fire, and assert the restriction that the `drink` for each course of this dummy meal must be a `KOSHER-WINE`.

The creation of dummy individuals must be performed with care, as CLASSIC assumes that they are distinct from all other individuals. Thus when the real meal is found, it cannot just be added, but, instead, either the dummy individual must be removed as a filler, or the two individuals must be merged in an application-dependent manner. It is best to use a dummy individual to answer the query, and then immediately remove it.

7 Conclusion

By now it is clear that learning a programming language involves more than just learning its syntax and semantics: there are usually an associated methodology or paradigm of use that needs to be absorbed, a collection of techniques for handling various special situations, warnings about frequent pitfalls, and the recognition that some other language might be more appropriate for a specific programming task. For example, in order to use PROLOG expertly one should, among other things, understand the paradigm of logic programming, the trick of building data structures with unbound variables (which are assigned a value later in the computation), the problems of negation by failure, and the cost of non-deterministic search/backtracking.

Knowledge representation languages are no different in this respect. For this reason, we have chosen to provide in this chapter more than just the description of an existing, implemented classification-based frame language. We have attempted to present the paradigm of using such languages by working through examples and by listing situations in which CLASSIC is likely to be useful. Additionally, we have indicated under what circumstances languages like CLASSIC may prove to be less than ideal. We have also assembled from our experiences of using the language and teaching it to others a collection of potentially confusing distinctions, together with "tricks of the trade" for representing special situations. Most importantly, we have presented a methodology for working through a domain and producing a knowledge base that reflects the domain structure in CLASSIC terms.

References

[Borgida *et al.*, 1989] Alex Borgida, Ronald J. Brachman, Deborah L. McGuinness, and Lori Alperin Resnick. CLASSIC: A structural data model for objects. In *Proceedings of the 1989 ACM SIGMOD International Conference on Mangement of Data*, pages 59–67. Association for Computing Machinery, June 1989.

[Brachman and Schmolze, 1985] Ronald J. Brachman and James G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, April–June 1985.

[Devanbu *et al.*, 1989] Premkumar Devanbu, Peter G. Selfridge, Bruce W. Ballard, and Ronald J. Brachman. A knowledge-based software information system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 110–115, Detroit, Michigan, August 1989. International Joint Committee on Artificial Intelligence.

[Devanbu *et al.*, 1990] Premkumar Devanbu, Ronald J. Brachman, and Peter G. Selfridge. LaSSIE—a classification-based software information system. In *Proceedings of the International Conference on Software Engineering*, Nice, France, 1990. IEEE Computer Society.

[Freeman, 1982] Michael W. Freeman. The qua link. In James G. Schmolze and Ronald J. Brachman, editors, *Proceedings of the 1981 KL-One Workshop*, pages 54–64, Jackson, New Hampshire, June 1982. Bolt Beranek and Newman Inc.

[Lenat and Guha, 1990] Douglas B. Lenat and R. V. Guha. *Building Large Knowledge-Based Systems*. Addison-Wesley, Reading, Massachusetts, 1990.

[Litman and Devanbu, 1990] Diane Litman and Premkumar Devanbu. Clasp: A plan and scenario classification system. AI Principles Research Department, AT&T Bell Laboratories, 1990.

[Nebel, 1988] Bernhard Nebel. Computational complexity of terminological reasoning in BACK. *Artificial Intelligence*, 34(3):371–383, April 1988.

[Owsnicki-Klewe, 1988] Bernd Owsnicki-Klewe. Configuration as a consistency maintenance task. In W. Hoeppner, editor, *Proceedings of GWAI-88—the 12th German Workshop on Artificial Intelligence*, pages 77–87. Springer Verlag, September 1988.

[Patel-Schneider, 1984] Peter F. Patel-Schneider. Small can be beautiful in knowledge representation. In *Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems*, pages 11–16, Denver, Colorado, December 1984. IEEE Computer Society.

[Peltason *et al.*, 1987] Christof Peltason, Kai von Luck, Bernhard Nebel, and Albrecht Schmiedel. The user's guide to the BACK system. KIT-Report 42, Fachbereich Informatik, Technische Universität Berlin, January 1987.

[Senyk *et al.*, 1989] Oksana Senyk, Ramesh S. Patil, and Frank A. Sonnenberg. Systematic knowledge base design for medical diagnosis. *Applied Artificial Intelligence*, 3(2–3):249–274, 1989.

[Yen *et al.*, 1989] John Yen, Robert Neches, and Robert MacGregor. Using terminological models to enhance the rule-based paradigm. In *Proceedings of the Second International Symposium on Artificial Intelligence*, Monterrey, Mexico, October 1989.