1986

Object-oriented programming Lisp Flavors and their application to a fire effects information system

James A. Mitchell
The University of Montana

Object-Oriented Programming, Lisp Flavors and Their Application to a Fire Effects Information System

by

James A. Mitchell

B.A., University of Colorado, 1976

M.A., Graduate Faculty, New School for Social Research, 1980

Presented in partial fulfillment of the

requirements for the degree of

Master of Science

University of Montana

1986

Approved by

Alden H. Wright

Chairman, Board of Examiners

[illegible]

Dean, Graduate School

November 17, 1986

Date

Mitchell, James A., M.S., November 6, 1986   Computer Science

Object-Oriented Programming, Lisp Flavors and Their Application to a Fire Effects Information System (169 pp.)

Advisor: Dr. Alden H. Wright AHW

An important feature of a developed expert system is its knowledge base. A knowledge base provides the factual and procedural information that expert systems use to make decisions within a specific problem domain. One approach to structuring a knowledge base is the use of frames within a semantic network. Frames act as information storing nodes that are connected by meaningful links. Traversal of these links results in a compilation of information, both factual and procedural, associated with a particular problem and its solution. An interesting feature of frames and semantic networks are their inheritance capability. Frames can be organized into a hierarchy of related information, with common information being stored at higher levels in the hierarchy. Frames that are lower in the hierarchy can inherit information stored at higher levels.

Frames, frame hierarchies, and frame inheritance all have a great similarity to the concepts associated with a newly popular artificial intelligence technique called object-oriented programming. In this project a description is provided of a conversion of an existing frame oriented knowledge base into an object-oriented one. The purpose of this conversion was to demonstrate that frame oriented systems are inherently object-oriented in nature.

Initially, an in-depth investigation of object-oriented concepts, their roots in data typing, and their developmental history, was performed. The existing frame oriented knowledge base, one belonging to a fire effects information system, was then decomposed into its component parts. Identified components included frames, frame hierarchies, frame accessing procedures, and frame inheritance. A direct mapping was then found between these frames concepts and the object-oriented concepts of the object, object classes, the message passing system, and inheritance capabilities, respectively. The investigation demonstrated that the existing knowledge base did have many object-oriented characteristics. The implications of using an object-oriented environment to build a knowledge base as opposed to creating a frame based one were then discussed and compared.

TABLE of CONTENTS

LIST of ILLUSTRATIONS

Chapter 1

INTRODUCTION

1.1 Project Background

This thesis project and paper is a direct result of the writer's participation in an experimental software development project, commissioned by the Intermountain Fire Sciences Laboratory (Fire Lab). The goal of this software project has been to attempt to utilize Artificial Intelligence (AI) techniques in the development of a Fire Effects Information System and Fire Prescription Expert System. It is planned that this Fire Lab project will span a period of five years. June, 1986, marks the end of the first year of this project.

The past year has been an important phase in the project's development, as the initial development period of any experimental software project is very crucial to later development. The decisions made at this stage greatly influence what is formulated later in the project. It is therefore very important that actions taken during this period in the project's development be well thought out. Additionally, since this software project is of an experimental nature, undergoing rapid evolution, the developers must be careful to build in a great degree of flexibility for future system changes and additions.

Complicating the system development requirements further is the fact that the majority of the software developers working on the project at

this time will not be with the project to its completion. Therefore, developed system components must be easy to understand and maintain.
As opposed to the normal type of software project, this research project is better characterized as one of iterative enhancement than as one fitting into the classical software development model. As each new feature and/or improvement is introduced into the system, it is as if a new system is developed. This process of iterative improvement makes it obvious that such a requirement for easy modification and maintenance of the information system requires the application of special software development techniques.

The proposed end goal of the Fire Lab software project is the development of a Fire Effects Information System and a Fire Prescription Expert System. By definition, such a goal requires the application of Expert System technology and thereby application of AI techniques. One important principle applied to the development of this initial information system has been the requirement that the developed database be later utilizible as a knowledge-base for the future Expert System. It is exactly this important principle which has led the developers to design and build the Fire Effects Information System using AI techniques. In particular, they have attempted to create an object-oriented frame-based system architecture to increase the ease of modification and maintenance.

1.2 Object-Oriented Programming Approach[1]

Object-orientation is a new approach to software development. It is a particular way of looking at the organization of data and procedures within a computer program. Instead of treating procedure and data as separate, as in standard programming, they are treated as a single unit called an "object". An object, therefore, is defined as a grouping of particular instances of data and the procedures that operate on that data. Operations upon these data are performed (procedures are invoked) by telling the 'object' (the grouping of data and procedures) the type of information that is wanted from it. The object is then responsible for performing the operation(s) upon itself and returning the desired information or result. These operations may return a value, set an internal value, calculate a value, or may perform any operation that has been defined to be performed on or with a given object.

For example, one might define a set of rectangles as individual objects. Let rectangle-1 have sides with lengths 3 and 5, rectangle-2 have sides 5 and 2, and let rectangle-3 have sides 10 and 8. Within most programming environments one would probably choose to represent each rectangle as a record or an array with each side being a field in the given record or an array index. One would obtain information about a given rectangle's characteristics, say its area, by retrieving the data in the side fields and then applying some procedure to those values to produce the value of its area. This requires that the programmer keep

[1]For a more complete description of object-oriented programming please see Chapter 2 of this paper.

track of where the data is, the type of values that are needed, and the appropriate procedures that can be applied to those values (i.e. a function that calculates the area of a rectangle and not that of a triangle).

Within an object-oriented environment, this bookkeeping is left up to the object itself and the programmer is free to concentrate on more abstract components of the program. Instance values are associated with the appropriate procedures which themselves know what values are needed to calculate the appropriate results. In the case of the rectangles, one would send a particular rectangle a message to retrieve its area. In one environment the call might be as follows:

```
(send rectangle-1 :area)
```

which would result in the value of 15 being returned. Doing the same with the other rectangle objects would result in values 10 and 80 being returned respectively.

The data contained within a particular instance of an object is often called 'instance data' and is held in 'instance variables'. In the example above, the actual values of the sides are instance values, and the side names would be instance variables belonging to each rectangle object (i.e. rectangle-1 would have instance variables side1 and side2 with instance data values 3 and 5). Procedures for operation upon this instance data are usually referred to as 'methods' (i.e. the rectangles would have associated with them a method called "area" that would calculate the required value using each rectangles' instance values).

The communication between the object and other parts of the programming system is usually called message passing (as seen in the call provided earlier to retrieve rectangle-1's area).

This technique of programming is particularly powerful as it allows the programmer and user to conceptualize system components at a higher level of abstraction. This abstraction also allows them to view components more like real-world objects. It also results in a hiding of procedural details, making programming of complex systems easier for the programmer and making program usage easier for the user.

Object-orientation also includes another important feature. Above, the word "instance" was used in describing data and objects. This is because within an object-oriented system characteristics of objects are described by an object descriptor. This is often referred to as a 'class'. Objects are organized into classes, and each class contains a description of the objects' characteristics and the procedures applicable to objects within that class. A particular object is an instance of a class. From the example provided above rectangle-1, rectangle-2, and rectangle-3 would be instances of the class 'rectangle'. Within the class description, objects' instance variables are defined along with the methods that can be applied to all the objects of the given class. For example the class 'rectangle' would contain information about instance variables 'sideA' and 'sideB' (used when creating a new instance), and methods for computing information (i.e. area computation). Individual objects of the given class may put values (instance data) into the instance variables, and utilize the methods defined by its class. These instance values are stored within the instance of the object, while the class level information is stored in the object class descriptor.

This again allows yet another higher level of abstraction for the programmer. By grouping objects into classes with the same characteristics but with different values for these characteristics, wholesale alteration and modification of all the objects within the class can be accomplished fairly easily by modifying the class descriptor. This greatly improves maintenance by centralizing the location of the procedural and descriptive information.

When applied to information system development, object-orientation requires developers to conceptualize information components as objects which themselves contain instance data and utilize procedural information about how to manipulate that data stored in some type of object class descriptor. Normal information systems may usually allow the grouping of data into entities, but restrict procedural information to external programs not directly related to the data object itself. When objects are changed, file structures and external programs must be modified, often drastically. Object-orientation seeks to avoid this problem by encapsulating object-specific data and procedural information into one package.

Object-orientation involves three main steps. First, the developer needs to create a means by which object characteristics can be described (instance values and value manipulation methods). He must also develop a system for describing meta knowledge about objects (i.e. class descriptors). Second, the developer must create a method for creation of instances of described objects. And third, the developer needs to create an interpreter, a message passing system, that can utilize these descriptions and instance values to retrieve information about the information objects within the system. In essence, this is exactly what has been done in the Fire Lab project.

1.3 Expert System Techniques[2]

Another important decision that expert system development requires is the choice of a knowledge representation for the information utilized by the expert system. The usual choice is between a totally rule based system, or a frame based system. A rule based system is one in which large amounts of procedural information is stored as a database of rules. This database is searched for applicable rules to be applied to a given state of information if certain conditions exist. The application of the rule(s) then produces a new information state which again utilizes the rule database.

---

[2]The following discussion is based on knowledge the writer has gleaned from coursework in Artificial Intelligence and from the following texts:

Charniak,E.,McDermott, D., Introduction to Artificial Intelligence, Addison-Wesely, Reading, Massachusetts, 1985.

Hayes-Roth, F., Waternam, D. A., Lenat, D. B., (Ed's), Building Expert Systems, Addison-Wesely, Reading, Massachusetts, 1983.

A frame based system is more like an information network, where each node in the network is a frame. A frames is somewhat like a record data structure. It is made up of a grouping of fields called slots. These slots contain information related to the frame. Each frame may therefore contain information about itself and its relation to other frames (nodes) in the network. It may also contain procedural information related to itself. In fact, slots might even contain rules to be executed by a rule interpreter. Information questions are answered by traversing this information network utilizing the information stored in the slots. This traversal might include application of rules or procedural information found in the slots of the frames. Frames can also represent hierarchies of information through their network connections to other frames.

A frame based system is more like the object-oriented system described above, where each frame can be treated as an object within the information network. In a rule based system, the given question would be transformed into an answer by the application of rules, while in a frame based system, it is answered by searching the information network for the information needed to answer it. This is similar to the retrieval of information from objects in an object-oriented environment.

In many expert systems, often a hybrid of the two methods is utilized. Totally rule based systems seem most appropriate when the data manipulated is small in comparison to the manipulations applied to it. In the case of the Fire Effects system the reverse seems more true of the system, little manipulation is performed on a large mass of data.

In this case, frames seem more appropriate and are what was chosen. This choice was made primarily due to the fact that a frame based system can be easily integrated with future rules and because it conveniently allows application of object-oriented techniques.

1.4 The Fire Effects Information System[3]

Within the Fire Effects Information System, frames form the basis of our object-oriented approach. The developers have created two major frame groupings. First are the actual data frames (class instances). These house the instance values (actual data) for each frame type (class) in the system. The system has many different types (classes) of data frames that represent the different information objects in the Fire Effects System. Second, are the system frames (object class descriptors). These frames contain descriptive and procedural information about frames of each type (these are class descriptor frames).

Another major component of the Fire Effects Information System is what the developers have called the interface functions. These functions act as the interpreter (the message passing and object creation system) that accesses and creates actual data frame instances, and utilizes the meta knowledge (class descriptor information) about data frames contained in the system frames. In addition, there are two external programs, a

[3]For a more in-depth discussion of the Fire Effects Information System please refer to Chapter 3.

knowledge base editor and a menu driven query program, that utilize these core components.

The objects of the system also have two more major features that have not as yet been described. First of all, the information is organized as a hierarchy of frames, with frames lower in the hierarchy containing more specific information about information in their parent frames. These form the different frame types of the system and the system's database structure. Secondly, the data frames have been broken down into groupings of lesser objects called slots. Slots represent each item of information contained within a frame. Like the data frames, each slot name (which may appear in different frame types) has a system frame that describes its characteristics and provides the procedural functions that may be applied to it. This again is an example of the direct application of object-oriented programming techniques, with data frames and slots corresponding to the instances of objects, system frames to class descriptors, and the interface functions corresponding to the message passing system.

One further feature of object-oriented programming that slots have that data frames do not is the addition of a higher level meta information descriptor frame (an object-class class descriptor). We were able to further group slots into five classes. System frames were created for each class, containing meta information that was common to slots of the same class. Slots utilize procedural and descriptive information stored here unless it is superseded by information in the slot descriptor system frame. This is an example of the object-orientation principles of object description hierarchies and property inheritance.

1.5 The Thesis

The previous discussion has briefly summarized what the developers have done on the Fire Lab project. They have applied a frame based object-oriented approach to the development of an easily modifiable information system. To do this they had to create an environment that implemented object-oriented programming constructs. But what if that environment already existed? Could they have accomplished the same end result? Or would they have had to implement an environment solely tailored to this particular application? It is this question that will be addressed in the remainder of this paper.

In the author's readings for this project he was introduced to four major object-oriented programming environments, namely Smalltalk, Loops, Objective-C, and Franz Lisp Flavors. Currently, Franz Lisp Flavors is the only conveniently available system to which this researcher has ready access, so the majority of his attention has been directed towards this implementation. Additionally, since the Firesys code is primarily written in Franz Lisp, it seems most appropriate to have focused upon this implementation of an object-oriented environment.

Franz Lisp Flavors appears to be an implementation of an object-oriented programming environment similar to that which was created for the Fire

Lab project. It is the premise of this thesis that it should be an easy task to convert the current Fire Effects Information System implementation into one utilizing Franz Lisp Flavors. This conversion was accomplished and has involved the reimplementation of the basic major components of the Firesys system in the Franz Lisp Flavors environment. The converted components included the database itself, the system meta-information database, and the interface functions. As hoped, it proved to be a fairly simple and straight forward endeavor. As a result of the conversion, knowledge regarding similarities and differences of the implementations, and answers to questions of the usability of such an environment with the Fire lab project were derived. This information will be discussed later in this paper.

In the following pages the writer presents a discussion of selected topics of interest related to this thesis project. The next chapter gives a detailed discussion of object-oriented programming in general, and a description of Franz Lisp Flavors and its relation to this programming technique. Chapter three provides a description of the Fire Effects Information System architecture and its relation to an object-oriented programming environment. Chapter four describes the Flavors implementation of the Firesys system. The final chapter discusses the success of the conversion attempt, similarities and differences between the implementations, advantages and disadvantages of the implementations, and whether there is any necessity for a custom environment.

Chapter 2

OBJECT-ORIENTED PROGRAMMING

2.1 Chapter Overview

Object-oriented programming is a newly popular and different approach to conceptualizing software program components [Alexander,1985] [Ingalls,1981] [Robson,1981]. Some computer science professionals think that the object-oriented approach will bring a revolution in programming during the 1980's like structured programming did during the 1970's [Rentsch,1982]. Languages that support it use concepts that attempt to increase the user-friendliness of programming and reduce the complexity that large programming projects often involve [Leiberman,1982] [Stoyan,1984]. These characteristics are accomplished by the introduction of two major concepts: (1) making problem solutions coded within computer programs more like solutions derived by human problem solving procedures, and (2) abstracting program components to a level that insulates the user and programmer from the implementation details [Alexander,1985] [Baroody,1981] [Ingalls,1981] [Sprague,1985] [Williams,1984]. These two concepts are closely related as the first cannot be accomplished without the second.

Object-oriented languages attempt to accomplish these characteristics by creating the concept of the 'object'. Objects are self-contained components that have values and behaviors. Like real world objects they can be manipulated, and based upon the manipulation will display certain behaviors. Such a modeling of real world objects is much more natural and simple to humans than standard programming concepts [Ingalls,1981] [Robson,1981] [Sprague,1985]. If computers are to assist humans by making tasks easier, then they should allow problem solving to be performed in the most human-like manner [Ingalls,1981]. Ingalls proposes that humans naturally classify and group elements of the environment as objects, and solve problems most naturally from this viewpoint [Ingalls,1981]. Object-orientation is also most natural because it mirrors the "subject-verb" orientation of the user [Ingalls,1981] [Sprague,1985] [Williams,1984]. Objects within the computer system therefore model how people perceive objects in the real world: they have identity, perform actions, may be grouped by similarities to other objects, and display actions and characteristics that are common to these groupings. It is conjectured that this approach results in the development of software products that are simpler to understand and maintain, that have shorter development times and greater flexibility, and that are more reliable [Cox,1984] [Ingalls,1981] [Pascoe,1986].

This chapter will attempt to demonstrate why these statements are true. First, a description will be provided of the object-oriented programming concepts. This will be followed by sections providing a brief history of object-orientation, its roots in the evolution of data types, its differences from traditional programming approaches, and some of its claims for software improvement. Finally, a description of the Franz Lisp Flavors programming environment will be given.

2.2 The Object-oriented Concepts

The object-oriented programming philosophy is composed of four primary ideas. First is the concept of the 'object' which is central to the whole approach. Second is the idea of message sending. Third is the hierarchical classification system. Lastly is the concept of inheritance. In this section, each of these concepts will be described.

2.2.1 The Object

The concept of the 'object' is central to the whole philosophy of object-orientation. Many definitions of the term 'object' are provided in the literature:

> Object: A package of information and description of its manipulations [Robson,1981].
>
> Objects have properties of 'objectness': inherent processing ability, message communication, and uniformity of appearance, status, and reference [Rentsch,1982].

> An object, far from being inert matter, is an active, animate entity, and is responsible for providing its own computational behavior. Its processing capability is not only inside the object, it is ever present within and inseparable from the object [Rentsch,1982].

> An "object" is like a package that describes a specific kind of data and the set of all procedures that may work on that data. Thus, an object is a higher-level grouping of information; a type of package designed for modularity and flexibility [Lubinski,1984].

> Object: The primitive element of object-oriented programming. Objects combine the attributes of procedures and data. Objects store data in variables, and respond to messages by carrying out procedures [Stefik,1986].

> An object consists of some private memory and a set of operations. The nature of an object's operations depends on the type of component it represents. A crucial property of an object is that its private memory can only be manipulated by its own operations [Goldberg,1983].

These definitions, in combination, describe the 'object' concept. An object is an abstract data entity, with hidden internal variables and values. Associated with these components are procedures (also called 'methods') which provide the only means by which the hidden values can be manipulated. Each of these data entity packages appear uniform from an external view, and can be accessed (invoked) only through the use of a standard message passing system (invocation protocol). This is the basic definition of an object.

Another important feature of the 'object' concept is the dichotomy of internal versus external view. Objects are always described as entities whose inner workings are hidden. This is no accident. The shift of viewpoint from the inside to the outside is in itself an essential part of the object-oriented approach. This shift allows for simplification of complexity, and allows programmers to conceptualize program components in a more natural way [Rentsch,1982] [Robson,1981]. Programmers can now utilize program components as they do objects in the real world. The programmer is only concerned with the inside view of an object when constructing the object itself. Once constructed, the internal details become immaterial to the object's usage. Only a knowledge of the messages that it will respond to is required [Rentsch,1982] [Robson,1981]. Internal implementations of objects can as a result be readily changed without affecting its interaction with other parts of the system as long as the message interface remains the same. This abstraction process and the ability to treat program components as objects are the real power of object-oriented programming.

2.2.2 The Message Sending System

The message sending system is also a primary concept of object-oriented programming. A user asks an object to carry out some action by sending it a message. The message sending system provides a means for activation of the object's operations to carry out a desired action. These operations are often called 'methods'. The object, upon receiving a message, carries out the associated action (method), returning the result that is needed. The object may not be able to carry out directly all of the action itself. It may have to send a message to another object which can provide the information needed to complete its task [Rentsch,1982]. Under such a system, instead of allowing procedures to access data structures freely, possibly causing unwanted side effects (as would be the case with the traditional procedurally oriented approach), one now has a system of objects (a union of data and procedures) cleanly passing information and carrying out actions via messages [Ingalls,1981].

Message sending is uniform. All processing is performed by sending messages. The same mechanism is used to do addition, file operations, and screen actions. This uniformity, like the uniform external view of an object, is claimed to simplify greatly the complexity of software systems [Rentsch,1982]. Uniformity of the invocation protocol (message sending system) supports the principle that calling programs should not make any assumptions about the implementation and internal representations of the objects they use [Stefik,1986]. It allows underlying implementations of objects to be altered without the need for changes to programs or other objects that call it [Stefik,1985].

Message passing is accomplished by sending an object an operation selector (also called a 'method selector'), useing a standard syntax.

Method selectors may be accompanied by additional parameters that might be needed for the called object to perform the desired task. However, a given method selector always will have the same uniformity (number of parameters) regardless of the object to which it is sent. This selector specifies <u>what</u> is to be done and not <u>how</u> to do it. It is left up to the receiving object to interpret the selector and to perform the requested action [Rentsch,1982] [Stefik,1985]. This message-sending paradigm along with the concept of the 'object' results in modularity by decoupling the intent of a message from the method used by the recipient to carry out the intent [Goldberg,1983] [Ingalls,1981]. These properties also insure that the implementation of one object cannot depend on the internal details of other objects, but rather only upon the messages to which they respond [Goldberg,1983]. It is claimed that this modular system structure may reduce the complexity of some software systems.

2.2.3 The Class System

The concepts presented so far describe the power that object-oriented programming provides with its modularity and uniform calling protocol scheme. But these advantages are not worth much if each object's internal code is a duplicate of the internal code of other objects of the same kind. If objects of the same kind really only differ by values in their internal state variables, then changes to the implementation of their operational procedures would mean making changes in every instance of that kind of object. Such a maintenance task would not be acceptable. The 'class' concept addresses this very problem.

Classification is an act that people do naturally every day. People abstract out those components of daily experience that are similar, and group those similarities in such a way that they denote the essence of those experiences [Cox,1984] [Ingalls,1981] [Rentsch,1982]. An example is the observation of a chair. When a person sees a chair, he/she does not only experience the chair as a singular object, but abstracts out of it the components that make it a chair like any other chair [Ingalls,1981]. Within object-oriented programming, the class serves a similar function [Ingalls,1981] [Rentsch,1982].

The class provides a description of all instances of objects in the class, much like a data type [Baroody,1981] [Robson,1981] [Stefik,1985]. It describes the implementation of a set of objects (its instances) that all represent the same kind of system component [Goldberg,1983] [Tyugu,1984]. The class provides a template for the creation of new instances by describing the form of their private memories (instance variables), and houses the operational procedures (methods) that are common to all of them [Goldberg,1983] [Robson,1981]. Each instance of a class contains instance variables whose contents describe their individual states. Additionally, they each have some name by which they can be identified as objects within the system, and some indication of the class to which they belong [Stefik,1985]. All messages sent to an object of a given class result in the application of the associated method (procedural code) stored in the class descriptor to the object's state values (if applicable) [Goldberg,1983]. This scheme allows for centralization of the code that is common to objects of the same kind. Additionally, introduction of new objects to the system only involves the creation of new instances of an already existing class. New classes can also be readily added if needed.

2.2.4 The Class Hierarchy and Inheritance

The existence of classes allows for code sharing and consolidation within an object-oriented system. Code that is common to objects of the same type can be factored out and stored in one central location for easy modification and extension. Objects of different types (classes) can then have the same message selectors, but belong to different classes. Each can have different implementations of the same type of actions. For example each object could be sent a 'print-self' message. Assume one of the objects is an integer, and another a string. Each would necessarily have a different procedure (method) to perform the print action. Because of the uniform message passing system and the class structure, all the objects could receive the same message ('print-self') and perform the correct action. Each object would access the needed procedural code from one location, its class. Objects of the same class (type) use the same code. But why stop there? There are certainly actions that are common to objects of different types (classes) that can utilize the same procedural code.

The concept of a class hierarchy addresses this issue. Classes may be broken up into a hierarchy of subclasses and superclasses [Goldberg,1983] [Robson,1981] [Stefik,1986] [Stoyan,1984]. Properties that are common to a grouping of differing objects can be centralized at a superclass level. For example, all motor vehicles have motors. A statement of this fact could reside in the superclass Motor_Vehicle. All cars and trucks when sent a message requesting an answer to whether they have a motor could access this method. Car and truck, being themselves separate classes, could have methods stored at their level that are unique to each of them. Likewise, car and truck themselves might have subclasses. Car might have subclass Compact_Car, or Mid_Sized_Car, each with special instance variables and methods.

The main concept here is that as methods and instance variables become more specialized, they reside in lower level classes in the hierarchy. More general ones are placed higher in the hierarchy. Lower level characteristics always override higher level ones. This results in a classification system that provides a spectrum of totally shared characteristics to totally individual ones [Rentsch,1982]. This kind of sharing makes for a usable system by factoring. Successful factoring results in brevity, clarity, and modularity, which in turn, it is claimed, results in manageability in complex systems [Rentsch,1982].

This class structure provides for adaptation by being variable along the dimension of individuality [Rentsch,1982]. What this means is that characteristics can be shared by the group while allowing individuals within the group to reinterpret some shared behavior as it applies to the individuals themselves [Rentsch,1982]. Allowing individual variability results in the capability of getting exactly what you want by overriding undesired group characteristics with individual characteristics [Rentsch,1982]. The hierarchy of classes specifically allows this to occur.

Object-oriented languages provide this capability to utilize or override grouped characteristics through inheritance [Robson,1981]. The idea here is that methods and instance variables defined at a subclass level will always override those defined at a higher level, otherwise the higher level characteristics become the defaults [Stefik,1985]. When an object receives a message it performs a bottom-up search of its class and superclasses to find the method associated with the received selector. The first one found will be executed, and will be the one with the correct level of specialization. This insures that procedures manipulate data at the proper level of abstraction [Baroody,1981]. Inheritance reduces the need to specify redundant information and simplifies updating and modification, since information can be entered and changed in one place [Bobrow,1986].

The power of inheritance is in the economy of expression that results from object description sharing [Stefik,1985]. This power is extended even farther by languages that permit 'multiple inheritance'. Multiple inheritance allows increased sharing by making it possible to combine object descriptions from many different classes [Stefik,1985]. Smalltalk, Loops, and Lisp Flavors provide these capabilities [Stefik,1985]. Each of these languages also provides a means for the user to specify some kind of precedence of inheritance from the multiple superclasses [Stefik,1985].

Object-oriented programming can now be seen as a different means of organizing and grouping program components. Fundamental to this approach is the creation of objects. Objects are packages of data and procedures with a uniform means of access. This uniform means of access is the same for all objects. Objects are organized into classes, similar to how humans organize objects in the real world. Common characteristics are abstracted to higher classification levels, and objects can inherit these characteristics if they belong to an appropriate subclass. Programs are created by establishing the appropriate objects, piecing them together, and having them interact with each other. This approach is reportedly more similar to how humans solve problems in the real world.

2.3 A Brief History of Object-oriented Programming

The immediate ancestor of all object-oriented programming languages is Simula where the class concept was introduced [Rentsch,1982]. However, Smalltalk still stands as the strongest representative of object-oriented programming in the sense of being the most unified in representing it [Rentsch,1982]. Awareness of the importance of object-orientation arose with the development of Smalltalk, so the history of Smalltalk is essentially the history of object-oriented programming [Rentsch,1982].

Smalltalk was originally the software half of a project called Dynabook, which was an effort to produce the most user-friendly computer [Rentsch,1982]. Alan Kay was the main visionary associated with this project, and in the late 1960's worked on a preliminary version called the Flex machine [Rentsch,1982]. Later in the early 1970's, he worked with others at Xerox Palo Alto Research Center (Xerox PARC) developing Smalltalk on the Xerox Alto machine [Rentsch,1982].

The development of Smalltalk drew heavily on the ideas of two older languages: Lisp and Simula [Rentsch,1982]. However, Smalltalk is primarily based upon the class concept borrowed from Simula [Rentsch,1982]. In Smalltalk the class is the sole structural unit, with instances of classes (objects) being the concrete units [Rentsch,1982]. Smalltalk is more than just a programming language. It is a total programming environment which reflects the object-oriented philosophy [Rentsch,1982].

Since the introduction of Smalltalk, awareness of object-oriented concepts has increased [Rentsch,1982]. Other languages incorporating object-oriented concepts have developed. These include: Lisp Flavors, Loops, Clascal, Objective-C, OOPC, C++, Neon, KEE, Object Lisp, STROBE, ACT I, Object Pascal, and others [Cox,1984] [Schmucker,1986] [Sprague,1985] [Stefik,1986] [Williams,1984]. The vast majority of these implementations, however, represent additions of object-oriented concepts to existing languages. This hybrid approach has been one aimed at trying to keep the best of both worlds [Cox,1984]. To the author's knowledge, Smalltalk still represents the only pure object-oriented programming language/environment [Rentsch,1982]. Due to the influence of the Smalltalk philosophy new machine environments have also developed. A prime example is the Apple Macintosh[1] computer with its object-oriented user interface which has borrowed heavily from research done at Xerox PARC and from Smalltalk [Sprague,1985].

One can see from the previous discussion that object-oriented programming has begun to attract much attention. Although its principal ideas have been around for some time, only lately has this great interest appeared. Introduction of object-oriented machines like the Apple Macintosh[1] may help to popularize this powerful programming paradigm, as may its application to existing programming languages and future applications.

2.4 The Evolution of the Data Type Concept

The evolution of the concept of 'data type' has played an important role in the development of programming languages [Pratt,1984]. The development of object-oriented programming marks a new stage in that evolution. It represents a new level of abstraction of data types beyond what languages based on other concepts provide. Object-orientation entails the optimal combination of the ideas of data encapsulation and data abstraction [Cohen,1984].

[1]The Apple Macintosh is a product of the Apple Computer Corporation.

Originally, computers were programmed using the memory locations of the hardware as the data object. Depending upon the context of its usage, that memory location could contain an integer, part of a floating-point number, a character, an instruction, or some other item. All data checking and usage was left to the programmer. Even though one can argue that specific instructions required data of a specific type, in actuality there really were no data types since no type checking occurred. Type conflicts were only evident when and if an error was identified in the programs behavior.

Older programming languages like FORTRAN and COBOL mark the beginning of the incorporation of the concept of a data type [Pratt,1984]. In these languages, primitive data types such as reals, integers, and character strings were provided. The compilers for these languages introduced type checking that insured that the programmer was utilizing them correctly. This early notion of data types centered around the concept that a data type defines a 'set of values' that a variable might take on [Pratt,1984].

The next level of evolution can be see in languages like Pascal [Pratt,1984]. In such languages 'type definitions' can be made that define the structure of a set of primitive data objects and their possible values. This allows the programmer to define a structured data type and to then declare instances of that type without having to redefine the whole structure for each instance [Pratt,1984]. At this stage the concept of a data type is expanded to mean a 'set of data objects and possible values'.

Pratt indicates that the 'final' step in the evolution of the data type concept is the understanding that a data type is not only a set of data objects and their possible values, but also a 'a set of operations' that manipulate objects of that data type [Pratt,1984]. With this he presents the idea of encapsulation. The idea of encapsulation is to have the programming language provide a means by which a data entity can be defined along with its data manipulations operations in a nice neat package, the internal details of which are hidden from the user of the entity. The manipulation operations provide the only means for accessing the data entity. These new data types are true data abstractions, leading to the concept of the 'abstract data type' [Pratt,1984].

The concept of an 'abstract data type' allows the programmer to abstract the complexity of a large programming project into smaller parts [Pratt,1984]. This allows the programmer to use effectively a 'divide and conquer' approach to the problem's solution [Pratt,1984]. Languages supporting these facilities include Ada with its 'packages' and Modula-2 with its 'modules' [Bobrow,1986] [Pascoe,1986] [Pratt,1984]. The two important ideas associated with this concept are (1) information hiding and (2) encapsulation [Pratt,1984].

Information hiding describes a central principal in the design of programmer-defined abstractions where each program component hides the details of its implementation from its user [Pratt,1984]. This suggests that each abstraction has a clearly defined purpose, and a specific interface through which the abstraction is manipulated. This kind of capability can be implemented in languages like FORTRAN by convention, but are not enforced by the language itself [Pratt,1984]. The addition of encapsulation capability (forced information hiding) by the language itself insures that later modifications cannot inadvertently breech earlier set conventions. Only languages like Ada provide such capabilities [Pratt,1984].

Pratt seems to think that data abstraction as he describes it is the "final" stage of evolution of the data type concept. The author does not believe this to be true, and neither do others [Buzzard,1985] [Pascoe,1986]. A language like Modula-2 allows the programmer to create abstract data objects through the use of the module (package) concept. Multiple instances of that data object can be defined as long as the named object is passed to its manipulation procedures. One problem arises when one wishes to change the abstract data type's composition only slightly, a whole new data type module must be reconstructed [Pascoe,1986].

For example, consider the definition of a stack data object. In Modula-2, a stack would be defined as an array or linked list of stack-type elements, and the operations push(), pop(), initialize(), empty(), and full(). However, the stack type definition would determine what type of elements could be put into the stack, say integers. To have another stack that allowed strings to be put into the stack would require that a whole new stack definition be created even though all but one line of code would be identical (stack_type = INTEGER versus stack_type = STRING) [Pascoe,1986]. The Ada concept of 'generic packages' attempts to address this issue, and will be discussed shortly..

There is an additional problem. We now have two modules with the same name! The compiler will not accept two definitions for the same object, 'stack'. So, we are forced to provide the different names, say String_Stack and Integer_Stack. Not only is this a problem with object names, but what happens when different objects have exported procedures (procedures declared to be accessible from outside the defined abstract object) with the same name? Take for example a stack and queue. Both probably need initialize(), empty() and full() procedures. If the names exported are the same, we have a problem. Their names must be unique or qualified (stack.initialize or queue.initialize) [Pascoe,1986]. The power of encapsulation and information hiding are present, but a major degree of flexibility is not.

What is needed is a new level of abstraction, and a new evolution of the abstract data type concept. Such an evolution is provided by the concepts of the 'generic package' and of 'operator overloading' seen in the Ada programming language [Buzzard,1985] [Pascoe,1986]. Generic packages allow multiple objects with similar but different structures to be created at compile time. This is accomplished by using a package template and checking the necessary type information [Pascoe,1986]. Ada also allows overloading of operators. Overloading makes it possible to have the same name for different but similar procedures. This capability eliminates the unique naming problem [Pascoe,1986].

But what happens if we want a structure that is not predefined at compile time, like a stack that can hold objects of different types? Such a capability requires dynamic binding [Pascoe,1986]. Ada attempts to address this problem with its variant records. Traditional programming languages can do this by providing some kind of case statement that checks types at run-time, applying the appropriate procedure for operating on a stack element of the given type. The problem here is that whenever a new stack element type is added to the system, not only is the code for the new type definition added, but the existing code (the case statement and variant record structure) for other objects (stacks) must also be altered [Pascoe,1986] [Winston,1981]. We now have a dependency between existing objects and new ones added to the system. Such a dependency defeats the encapsulation we have strived for by requiring knowledge of the implementation of all the data objects in the system!

Again, we need another evolution in our concept of an abstract data type. This evolution involves the addition of the concept of the data object as being an animate object. In this abstraction, the object itself becomes responsible for performing operations on itself, no longer being dependant upon external procedures [Pascoe,1986]. This eliminates the need for the case statement mentioned in the stack example previously, as now the stack element itself would perform the operation.

But we still have the problem of having redundant code for highly similar operations. A slight modification in the behavior of an operation will involve alteration of all the code for the similar operation. As noted earlier, Ada provides the generic package [Buzzard,1985]. In a way this is really a form of inheritance [Rentsch,1982]. Each instance of the generic package inherits the characteristics of the generic package with minor modifications. However, inheritance is limited to one generic package. There is no hierarchy of inheritance.

This idea of inheritance is the next level of abstraction that is brought to programming by an object-oriented approach. Inheritance allows code to be factored [Pascoe,1986]. Code that is common to data objects can be stored in one location. This, it is conjectured, makes modification of code easier and more reliable [Cohen,1984]. Factoring is accomplished by defining classes. Classes can have subclasses or superclasses. Common code can be stored within these class definitions, dependent upon their level of factoring [Pascoe,1986].

The evolution of data types described to this point now includes quite a few more characteristics than those Pratt [Pratt,1984] has described in his "final" stage. We now have arrived at a description of an abstract data type as an 'object'. This 'object' is a set of data objects (abstract types or values) with procedures to operate on itself, with encapsulation of these components resulting in information hiding, with inclusion of dynamic binding and class inheritance capability, and with the inclusion of the concept of an 'object' as an animate entity [Pascoe,1986] [Stefik,1986] [Stoyan,1984]. The application of this abstraction to programming supposedly results in software that is more flexible; supporting change, reusability, and easy enhancement [Cox,1984].

2.5 Traditional versus Object-oriented Programming

As mentioned in the beginning of this chapter, object-oriented programming is a different approach to programming. Different as compared to what? This section will describe the differences between what is called traditional or procedure-oriented programming and object-oriented programming.

The traditional or procedural-oriented style of programming can be described as dividing programming into two distinct segments [Cox,1984]. First is the code segment, consisting of subroutines that do all the work of the program. Second is the data segment, consisting of the data structures that the procedures manipulate [Bobrow,1986] [Cox,1984] [Robson,1981]. Data are static, having values changed by procedures, and are essentially global [Cox,1984] [Leiberman,1982] [Stoyan,1984]. Major operations are built by combining subroutines into sequences that are grouped [Cox,1984]. Procedures are responsible for keeping track of timing considerations (sequence), space and movement of data, and data type checking [Cox,1984].

One problem with the procedure-oriented approach is that data and procedures are treated as if they are independent of each other when in fact they are not [Cox,1984] [Robson,1981]. Procedures, in practice, place strong restrictions upon the types of data that they handle [Cox,1984]. This fact results in the need to do major surgery to general-purpose procedures when changes are made in data structures or when new data structures are added [Pascoe,1986] [Winston,1981]. The procedure-oriented approach makes the programming environment responsible for managing data type dependencies, so environmental code is not reusable [Cox,1984]. Additionally, the programmer must remember what these restrictions are when using the procedures and this results in errors being made [Cox,1984].

An interesting example is provided by Cox [Cox,1984]. What would we think if an electrician who was wiring telephone lines and power lines in a building was required to use the same type of plugs and wires to do both? It would be his responsibility to remember which plug was carrying what voltage! This is the situation created when using procedure-oriented programming techniques; we attempt to keep track of compatibility information manually [Cox,1984].

The object-oriented approach, in contrast, treats procedures and data as two indivisible aspects of the same object in the problem domain [Cox,1984] [Robson,1981]. Applications can be developed by straightforwardly examining the problem domain, identifying objects and their behaviors within the domain, and then implementing them in the computer utilizing object-oriented techniques [Cox,1984]. The programmer is no longer required to restate the problem domain into computer domain terms where everything is either an operator or an operand [Cox,1984]. No longer is knowledge of data characteristics spread through all the procedures of a program, but rather centralized to specific data objects [Bobrow,1986] [Leiberman,1982]. Each object has only the knowledge and expertise to act in accordance with requests made of it, placing knowledge only where it is actually used [Leiberman,1982]. Data/procedure interdependencies are moved out of implicit storage in the environment and into explicit storage in the data objects themselves [Cox,1984].

As opposed to function calls with static-data passage, object-oriented programming utilizes a message-passing system [Bobrow,1986] [Cox,1984] [Leiberman,1982] [Robson,1981]. An object is sent a message and responds to that message according to its internal knowledge. Like function calls, messages can contain parameters. The object determines how to perform the action itself [Robson,1981].

Another important difference is the ability that object-oriented programming has to factor common code out of the object's local structure, placing it into a common location [Bobrow,1986] [Cox,1984] [Leiberman,1982] [Robson,1981]. Objects are defined by their class. A class, in turn, can be described by another superior class. When a message is sent to an object an upward search is performed within the class hierarchy structure for a procedure that matches the message request. If none is found and no superclasses remain, then an error message is issued [Bobrow,1986] [Cox,1984] [Leiberman,1982] [Robson,1981] Code that is common to several classes is stored higher in the hierarchy. This technique of code factoring, called inheritance, is a scheme that allows new objects to be easily added to the software system without major modification, since new classes can easily be defined by declaring them as subclasses of existing classes [Bobrow,1986] [Cox,1984] [Leiberman,1982] [Robson,1981].

These differences give object-oriented programming some advantages over procedure-oriented techniques. Data dependencies encoded within procedures are eliminated. Code modifications and additions are made simple and side effects are minimized. Programmed problem solutions are not forced into computer defined structures (i.e. the data types available), but rather allow abstract data object definitions that parallel real world problem domain structures. Code factoring and compression are also a natural part of this programming style. Because of these differences, object-oriented programming may be an important and powerful improvement over traditional programming techniques.

2.6 Why Object-oriented Programming?

In the previous sections, the reader has been presented with the basic concepts of object-oriented programming. Additionally, the reader should now be familiar with the basic history of the development of object-orientation, and its difference from traditional programming. But why should the user utilize this programming technique? In this section, some of the claimed benefits of object-oriented programming will be presented. Object-orientation's relationship to software cost and maintenance will also be described. Lastly, a description of some programming projects to which the technique was applied will be presented.

2.6.1 Some Claims of Object-oriented Programming

The Fifth Generation of computing has been heralded as being at hand due to the new advances in Artificial Intelligence (AI). Associated with this evolution are at least three developments in software technology: logic programming, exploratory programming, and object-oriented programming [Sheil,1983]. Based upon statements like this one might claim that object-oriented programming is a new and revolutionary AI technique. This is apparently due to the close relationship that object-oriented programming has with the theory of frames [Barbuceanu,1984] [Stefik,1985].[2] Others have claimed its usefulness for simulation programming, systems programming, and graphics [Bobrow,1986] [Stefik,1985].

With regard to simulation, objects can form the basis for simulation of system components and their interactions. Conceptualizing system components as objects reportedly makes simulation programming conceptually easier [Barbuceanu,1984] [Ingalls,1981] [Stefik,1985]. In general usage, large classes of computer applications attempt to model some physical or conceptual process. Traditional programming makes the programmer force this modeling into some machine representation that is

---

[2] A discussion of object-orientation's similarity to frame theory will be presented in Chapter 3.

often not in a form parallel to the real world process. Object-orientation, on the other hand, by design, models real world objects and events, and parallels conceptual processes, making it better for simulations and any other form of modeling [Cox,1984].

2.6.2 Software Cost and Maintenance Considerations

By far, software has become the most costly portion of most computer systems [Lubinski,1984] [Martin,1983]. According to James Martin [Martin,1983], sixty-seven percent of that cost can be accounted for by maintenance needs. With this fact in mind, one is faced with the necessity of making software as easy to understand and maintain as possible. A primary feature of object-oriented programming is its inheritance and classification capabilities [Alexander,1985] [Alws,1985] [Brown,1983] [Cox,1984] [Goldberg,1983] [Leiberman,1982] [Lubinski,1984] [Rentsch,1982] [Stefik,1985]. These capabilities allow code that is common to different types of objects to be stored in one location that is accessible to all of these objects. If an object belongs to a classification, it can inherit any code that is associated with that classification. This makes for the elimination of redundant code, allowing code sharing and centralization. Code maintenance and modification then should become much easier, because the code is more compact and centralized. Cox [Cox,1986], suggests that what is truly revolutionary about object-orientation is that it helps programmers to reuse existing code. He offers as an analogy a comparison of object-oriented programming with circuit building using IC-chips (Integrated Circuit chips). He suggests that objects in object libraries are "Software-ICs" [Cox,1986]. The results of reusability can be seen if one compares the size of the Unix operating system (non-object-oriented) with that of Smalltalk (totally object-oriented). One finds that on a capability based comparison, Smalltalk has much less code than Unix [Cox,1984]. This reduction is reported by Cox [Cox,1984] to be due to Smalltalk's centralized and shared code. However, one should temper this statement with the knowledge that Unix may provide a greater number of system capabilities.

Additional important features of object-oriented languages include its object modularity, and uniformity of invocation protocol [Alws,1985] [Brown,1983] [Cox,1984] [Goldberg,1983] [Ingalls,1981] [Leiberman,1982] [Lubinski,1984] [Rentsch,1982] [Stefik,1985]. These factors also directly affect the maintainability of a software system. By definition, objects are encapsulated units, containing values and procedural information with a uniform interface. This structuring insures that implementation details of object structure and behavior are totally hidden from the object user, thereby eliminating environmental dependencies that might otherwise reduce the flexibility of the software [Cox,1984] [Goldberg,1983]. Objects are self-contained entities that can only be examined externally, and whose internal workings have no dependency on external conditions. Languages like Ada also attempt to meet this high degree of maintainability through the concept of the package, but lack the class hierarchy and uniform invocation protocol capabilities of object-oriented languages.

Even though the concept of an object as a self-contained entity is powerful, its true power is not realized until one recognizes the importance of the concept of a uniform invocation protocol [Goldberg,1983] [Ingalls,1981] [Rentsch,1982] [Stefik,1985]. Values are retrieved and procedures invoked by passing a message to an object. All objects can receive any message, and will respond in one of two ways. Either the object will do the desired task, or it will notify the caller that it cannot perform the task (send back an error message). The real power here is that at any time an object can be added or removed from the system without requiring the alteration of existing system code. Because the message passing system is uniform, only the code for the object in question need be affected [Goldberg,1983] [Ingalls,1981] [Rentsch,1982] [Stefik,1985].

Object-oriented programming may greatly enhance the maintainability and flexibility of software. As noted above, common code can be shared and centralized, objects are encapsulated eliminating external dependencies, and invocation of object actions is uniform. These characteristics, it is claimed from programming experience, make object-oriented code highly reusable, and easier to maintain and modify than programs coded with traditional techniques [Alws,1985]. These features are also claimed from experience to support dramatically the ability to perform rapid prototyping [Alws,1985]. Object-oriented software development techniques therefore show promise for providing an environment in which programs can be developed modularly, with a minimum of inter-module coupling (dependency), and with the flexibility to be easily maintained and modified.

2.6.3 Object-oriented Applications

Currently, the use of object-oriented techniques is open to much experimentation and many different environments have been created to date [Stefik,1985]. Within these environments different application programs have been developed. One such application was constructed at Tektronix Inc. using Smalltalk (the prototypical object-oriented programming environment [Rentsch,1982] [White,1986]) [Alexander,1985].

Tektronix has the difficult task of diagnosing and repairing electronic equipment that it sells. Training technicians to have a concise and highly developed fault isolation strategy is very costly and time consuming. Additionally, once trained, many technicians soon move on to new jobs. This situation makes electronics troubleshooting an ideal application for an expert system. Tektronix decided to create a technician's assistant to help assist and guide technicians in repairing equipment [Alexander,1985].

The task involved the conceptualization of electronic components as objects in the software system. Each object was coded to display behaviors that were expected of their real world counterpart. Utilizing the outstanding graphics of Smalltalk, circuit diagrams and components could be displayed as part of a diagnosis simulation [Alexander,1985]. The program presents a display showing the circuit diagram and board layout for the component to be tested. Expected voltage readings for pointed-to components within the display are shown, allowing anomalies to be easily recognized when comparisons are made to actual readings. If the technician requests diagnostic assistance, the program queries for circuit readings and additional information, and suggests a new course of action for the technician to take [Alexander,1985].

The user is led through the diagnosis process by the program, not only assisting him in the task, but actually training him in a diagnosis strategy. The Smalltalk object-oriented environment with its 'objects' and hierarchical classification capability has allowed such a simulation to be coded with a minimum of effort and with maximum flexibility. Each assistant for different electronic equipment was coded using the same base program [Alexander,1985].

Smalltalk is not the only language used for object-oriented application development. OOPC (Object Oriented Precompiler for C) has also been utilized [Cox,1984] [Awls,1985]. In the Awls implementation [Awls,1985], two special purpose editors were developed. The editors were designed to assist software designers in producing documentation for designs for software projects. One editor was constructed to build special system structure charts, and the other to develop pseudo-code for designed modules [Awls,1985]. Modules designed were treated as objects that needed to be represented by diagrams and pseudo-code by the editors. According to Awls, object-oriented concepts allowed the editor programs to assist the designers in keeping track of module interfaces and procedural interactions [Awls,1985]. This anecdotal program description suggests that object-oriented techniques can assist project developers with integration of disparate project components.

Written in a special language called Act 1, Leiberman has constructed a composers assistant [Leiberman,1982]. The program is utilized by musicians composing music. Notes, chords, and melodies make up the objects of the system. The program can be used to analyze existing compositions, or to assist in creating new ones. Leiberman states that traditional programming languages are not very good at dealing with the complexity that a task such as music composition entails, and that object-orientation is one approach that makes the complexity easier to handle [Leiberman,1982]. His experiences with utilizing object-oriented techniques lend support to the notion that they reduce project complexity.

Other applications have also been constructed using object-oriented programming techniques. They include: (1) a Computer-Aided Design (CAD) system that intelligently simulates design activities, illustrating design consequences [Barbuceanu,1983], (2) the repackaging of a Graphical Kernel System so that it is easily accessible by applications in the most flexible manner [Lubinski,1984], (3) development of a highly flexible multi-user database system with easily customized user interfaces [Baroody,1981], (4) creation of an electronic form handling system that updates and manages forms used in planning and arranging executive business trips [Fikes,1981]. All of these applications lend support to the great potential that object-oriented programming holds for computer software systems.

2.7 Franz Lisp Flavors[3]

Flavors is a name for a more general class of object-oriented extensions to a Lisp dialect. It is not specific to the Franz Inc. version of Lisp. The object-oriented style implemented in Franz Lisp Flavors is borrowed directly from the Smalltalk and Actor families of languages. The Franz Lisp implementation of Flavors is similar to Zetalisp.

Flavors is an extension to Franz Lisp in the sense that it utilizes the hybrid approach mentioned earlier, taking a standard Lisp implementation and adding new object-oriented capabilities to it. Therefore, Flavors is not a totally object-oriented programming environment, but rather an enhancement of an existing Lisp language.

With regard to this thesis project, the usage of Franz Lisp Flavors is most appropriate. The original Fire Lab Project code was written in this dialect of Lisp and any conversion of the Fire Lab code into a standard object-oriented form could be accomplished in a straight forward manner using this extension. This is exactly the reason that Franz Lisp Flavors was chosen for the language of implementation of this thesis project.

---

[3]All information regarding Franz Lisp Flavors presented in this section has been taken directly from Chapter 19 of the Franz Lisp Reference Manual, Franz Lisp Opus 42.16.3, Franz Inc., 1985.

Although it can be argued that usage of a hybrid approach in creating an object-oriented programming environment is in opposition to object-oriented precepts, hybrid languages allow the usage of existing programming techniques and code which can be enhanced with new and powerful programming techniques [Cox,1984]. In the case of the Fire Lab Project, a large mass of Lisp code was already in existence, and the author was familiar with the Franz Lisp language. Additionally, it was the purpose of this thesis project to demonstrate that the project team had actually created a custom object-oriented environment. Usage of an object-oriented extension to Franz Lisp fits this purpose perfectly.

Franz Lisp Flavors provides all of the capabilities described in the previous section of this chapter. It allows object instances, classes, methods, and class hierarchies to be created. As noted above, it also allows the creation of class hierarchies that are not restricted to a tree structure. Rather, Flavors allows a graph structure (multiple parents), which in turn allows arbitrarily complex interconnections between object classes while retaining modularity and ease of maintenance [Brown,1983]. In the following sections, a brief description of Franz Lisp Flavors syntax and capabilities will be provided.

2.7.1 Franz Lisp Flavors Objects

An object in Franz Lisp Flavors is created much like objects described earlier. First, a class must be created, and then instances of that class are formed. In Flavors a class is called a 'flavor'. To define a flavor (class), one uses the 'defflavor' function:

```
(defflavor ship (x-position y-position
                 x-velocity y-velocity mass)
                ()
                :inittable-instance-variables
                :gettable-instance-variables
                :settable-instance-variables)
```

This construction defines a flavor (class) called 'ship' that has five instance variables that specify a ship's position, velocity, and mass. As can be seen the definition specifies that these variables can be externally retrieved and set. Instance variables can also be initialized with values. To create an instance of a ship, we must create a name for the object, and call a function to make an instance:

```
(setq my-ship (make-instance 'ship))
```

As one who is familiar with Lisp syntax can see, this form is in normal Lisp syntax. It is not as one would expect if the environment were totally object-oriented. In such an environment, a message would be sent to the class 'ship' to produce a new instance, and an assignment would be made to a specified name with the returned object. In this case, exactly the same action is performed, but with normal Franz Lisp syntax. In any case, the result is an object named 'my-ship' that has the instance variables described in its flavor (class) 'ship'. If one wishes to initialize 'my-ship's variables the syntax would be as follows:

```
(setq my-ship (make-instance 'ship
                             :x-position 0.0
                             :y-position 2.0
                             :mass 3.5))
```

This form would produce 'my-ship' with position (0.0,2.0) and mass 3.5. Values can also be initialized for all instances by including values within the flavor definition itself:

```
(defflavor ship ((x-position 0.0)
                 (y-position 2.0)
                 x-velocity
                 y-velocity
                 (mass 3.5))
          ()
          :inittable-instance-variables
          :gettable-instance-variables
          :settable-instance-variables)
```

In this example, all 'ship' instances would start off with position (0.0,2.0) and mass 3.5. The velocity values would remain as yet undefined.

2.7.2 Franz Lisp Flavors Messages

The message sending facility provided by Franz Lisp Flavors is also more in the syntax of Franz Lisp than in what would be expected in a totally object-oriented programming environment. In a language like Smalltalk, a message is sent by following an object name with a selector [Goldberg,1983]:

```
my-ship mass.
```

This Smalltalk statement would send 'my-ship' a message to return the value of its mass. In Franz Lisp Flavors the 'send' function is utilized to transmit messages to objects. Its syntax would be as follows:

```
(send my-ship :mass)
```

Again, this form would send the message 'mass' to 'my-ship', and the value 3.5 would be returned. All message-sending is done with this function. To change the mass of the ship, a message like this could be sent:

```
(send my-ship :set-mass 35.5)
```

In this example, the method (object manipulation procedure) :set-mass has a parameter. Methods like :mass and :set-mass are already predefined by the Flavors system when an instance of a 'ship' is created.

2.7.3 Franz Lisp Flavors Methods

So far the Flavors object definition capability and message passing system have been illustrated. But messages need methods (procedures) associated with them. As noted above, instances have predefined methods which allow the retrieval and setting of instance variable values. These are methods that belong to the flavor 'vanilla'. Vanilla provides additional methods: :print-self, :describe, :which-operations, and several others. All Franz Lisp Flavors objects include the 'vanilla' flavor. However, there is no real power to Flavors if one cannot define his/her own methods.

Franz Lisp Flavors provides the 'defmethod' function to create methods for objects. As in other object-oriented languages, methods must be attached to the objects class. In this case, the method is associated with a flavor:

```
(defmethod (ship :speed) ()
   (sqrt (+ (^ x-velocity 2)
            (^ y-velocity 2))))
```

This Franz Lisp form defines a method named ':speed' that is associated with the flavor 'ship'. The method will take the velocity instance variables of the object it is applied to and calculate the velocity (creating a vector using the x,y velocity components). Methods can also be defined that utilize parameters:

```
(defmethod (ship :fraction-of-speed) (fraction)
    (* fraction (send self :speed)))

(send my-ship :fraction-of-speed .5)
```

This method definition uses the parameter named 'fraction', and multiplies it by the calculated speed of the ship to which the method is applied. The message example would return a speed value that is one half the actual speed due to the parameter value of '.5'.

Please take note of a special feature illustrated in the :fraction-of-speed method definition. Within the method definition there is a message sent to 'self'. While any method is executing, the variable 'self' is bound to the identifier of the object to which the method was applied. This allows a method to call other same flavor methods during its execution. In the above example, the calculation of the speed is performed by another method, which returns the value needed to complete the fraction calculation.

Messages can also be sent to another object during method execution if the other object's identifier is passed as a parameter:

```
(defmethod (ship :collision) (object)
    (intersect (send self :direction)
               (send object :direction)))

(send my-ship :collision your-ship)
```

Assuming that there is a function 'intersect' that can calculate if two objects will intersect given their directions, the above method definition would provide the message-sender with the knowledge of an impending collision.

2.7.4 Franz Lisp Flavors Classification Hierarchy

Within Franz Lisp Flavors, a class hierarchy is defined by mixing flavors. Flavors are mixed by providing the identifiers for the 'mix-in' flavors in the flavor definition:

```
(defflavor ship (x-position y-position
                 x-velocity y-velocity mass)

                (moving-object)

                :gettable-instance-variables
                :settable-instance-variables)
```

In the example, 'moving-object' is identified as a 'mix-in' flavor. All instance variables and methods that belong to 'moving-object' are included (referenced by) the 'ship' flavor unless overridden by local 'ship' specific variables or methods. This structure in essence is a specification of 'ship' as a subclass of 'moving-object'. The 'ship' class of objects <u>inherits</u> the characteristics of the 'moving-object' class unless locally overridden.

As noted earlier, Flavors has the capability to allow multiple parents (multiple hierarchies). It does this by allowing multiple mix-in's:

```
(defflavor ship (x-position y-position
                 x-velocity y-velocity mass)

                (moving-object
                 floating-object
                 sinking-object)

                :gettable-instance-variables
                :settable-instance-variables)
```

Here, 'ship' now inherits the characteristics of 'moving-object', 'floating-object', and 'sinking-object'. This could become very confusing if there were no way to define an order of inheritance. Franz Lisp Flavors defines such an ordering of inheritance by specifying that the order of mix-in's matters. The inheritance proceeds on a depth-first search of mix-in's in the left to right order of the mix-in list.

Mix-in's themselves are also flavors. They too can be made up of other mix-in's. In this way a graph or network structure of inheritance can be constructed. However, within such a network there is always a potential for cycles to occur. The Flavors language extensions take care of this by not allowing the method search to cycle. No flavor node in the graph can be visited more than once. All flavors also include the flavor 'vanilla'. Vanilla flavor provides some basic methods that all objects may need. Vanilla flavor can be left out if so specified in the flavor definition.

The preceding discussion has introduced some of the basic features of Franz Lisp Flavors. As one can see, all the basic object-oriented capabilities expected in an object-oriented programming environment are present. However, some of these capabilities are not provided in syntactic forms that are totally consistent with an object-oriented philosophy (making an instance for example). Even so, the provided capabilities are very powerful and in some cases go far beyond what other environments provide.

The descriptions presented here have been of an introductory nature. Franz Lisp Flavors provides many additional features that have not been covered. Interested parties should refer to the Franz Lisp Reference Manual[3]. Experimentation with a Franz Lisp Flavors implementation is highly advised.

Chapter 3

THE FIRESYS PROJECT

3.1 Firesys Project Goals

Initially, the intended goal of the Firesys project was to develop two expert systems. The first system to be developed was a fire effects advisor. The second was to be a fire prescription expert. The two systems were to share a common knowledge base, and were to be initially restricted to providing information regarding sagebrush ecosystems.

The fire effects advisor was to provide the system user with answers to questions about the effects of fire. Sagebrush range managers often need fire effects information to assist them in making decisions regarding the use of fire as a range management tool. The information needed includes both the short and long term effects on plant growth, wildlife forage, and cover. Once a decision to utilize fire for management of a specific site is made, a fire use prescription is then needed. The second expert system was to provide such a prescription. The user would provide goal and site descriptions, and the system would provide a prescription for the type of fire and conditions needed to attain the desired goal.

3.2 The Initial Effort and Resulting System[1]

One of the primary tasks that expert system builders face is the decision on how to structure the knowledge base used by the expert system. The choice of a knowledge base structure is the primary determinant of the expert system's later capabilities since system actions and structure are determined directly by the knowledge base. As noted in chapter 1, there are two common approaches to knowledge base design. One can encode knowledge in the form of rules or as frames. Mixtures of the two can also be utilized.

Rule based or production systems normally use a retrieve-act cycle. The expert system retrieves a rule from the knowledge base dependent upon the system's current state of information. It then applies the rule to its information state (the state-record), changing it. This action continues until the desired state (goal) is reached, or until no rules can be found that apply (failure). Rules, therefore, usually have the following form:

<IF state THEN action>

[1]The following discussion of expert system knowledge bases is based on information the writer has gleaned from coursework in Artificial Intelligence and from the following texts:

Charniak,E.,McDermott, D., Introduction to Artificial Intelligence, Addison-Wesely, Reading, Massachusetts, 1985.

Hayes-Roth, F., Waternam, D. A., Lenat, D. B., (Ed's), Building Expert Systems, Addison-Wesely, Reading, Massachusetts, 1983.

where the rule is chosen if the system's state conditions match 'state'. The 'action' of the rule is then applied to the system's state conditions stored in the state-record, changing them in some way.

```
Example Rule:      IF blood test negative AND
                      urine test positive
                   THEN test thyroid level AND
                        add to state-record
```

The cycle is then repeated using the new state information. The system's initial state might have a statement of the goal to be reached (question to be answered) and the starting givens. Because the rules essentially manipulate the initial state of the system into a desired state through actions, one can see that such a technique is best applied to tasks that involve large amounts of procedural as opposed to factual knowledge.

Another common rule based approach is to use what is called 'backward chaining'. Under this method the system starts with the goal state and attempts to verify that rules and facts in the knowledge base allow one to conclude that the goal state is true. The method works much the same as the above described except that rule conclusions are utilized. The backward chaining system examines knowledge base facts and rule conclusions to see if they match the goal state. If a fact matches then the goal has been verified to be true. If a rule conclusion matches, then the system attempts to verify that the rule antecedents can be verified. The rule antecedent(s) become the new goal(s) to be verified. A backward chaining rule commonly has this type of structure:

```
<conclusion IF antecedent>

Example Rule:     'Sunny Outside' IF 'Day Time' AND NOT 'Cloudy'
```

The backward chaining process continues until the goal is verified to be true, or until no facts or rules remain as verification candidates.

The opposite approach to rules is that of a frame based system. In such a system, a semantic network of knowledge is constructed. Each node of this network is a frame. A frame contains information related to itself and about connections to other frames (nodes). The connection information is also encoded so that it expresses the frame's relationship to other nodes. Frames usually have the following structure:

```
<ATTRIBUTE-1 trait-1
 ATTRIBUTE-2 trait-2
            .
            .
            .
 ATTRIBUTE-n trait-n>
```

where an attribute is a characteristic of this node or a name of a connection or relationship to another node. Traits are therefore facts about the attribute or names of (pointers to) other frames (nodes). When one speaks of frames, attributes are usually called "SLOTS" and traits "SLOT FILLERS". The following example frame might describe a specific dog:

```
Example Frame:    <NAME "Fido"
                   COLOR      blond
                   IS-A dog
                   SIZE medium
                        .
                        .
                        .
                   OWNER      Sam>
```

In the above example "Fido", 'blond', and 'medium' are specific facts about the dog, and the remaining traits (slot fillers) are names of other frames that further define characteristics of the "Fido" frame. The frame "dog" would provide information about dogs in general, such as body parts, while the frame "Sam" would describe the owner's characteristics. This type of frame structure allows a large amount of facts and their interrelationships to be encoded into a knowledge base. Tasks that involve the gathering and assessing of large amounts of factual knowledge are therefore best handled with an expert system that utilizes frames.

As noted above, one can construct a system that uses a hybrid knowledge base. Rules can include factual information that can be added or deleted from the state-record. Frames can contain attribute fields that have procedural information (actions) as traits. For example, in the "Fido" frame above, we might add an attribute like IF-BITES-KIDS with the trait value 'get rid of Fido and remove from network'. In this way rule-like procedural knowledge can be added to a frame, or frame-like factual knowledge can be included in a rule. In general, this is often how expert system developers deal with tasks that require combining factual and procedural knowledge.

The first Firesys project developed, the fire effects advisor, was an expert system which required the storage of large amounts of factual information upon which smaller amounts of procedural information were to be applied. The majority of the encoded knowledge was to be factual knowledge about plant species and data on effects of fire on each species as extracted from the research literature. The system was to sift through the data, analyze the facts related to the management objective provided by the user, and provide some conclusion as to whether the objective would be met. This task requirement made it obvious that a frame based expert system would be most appropriate, so the decision was made to adopt this approach.

As development of the fire effects advisor progressed, the focus of effort became more and more directed towards the encoding of the factual knowledge. Procedural knowledge became less emphasized due to the enormity of the fact-gathering task. Additionally, the purpose of the system was reformulated, playing down the analysis capability, and emphasizing information retrieval. The system was now to be more of a research aid, or on-line library, for managers to use for gathering facts for their analysis of management objectives. The objective of the fire effects advisor was now to provide information, and not advice.

Thus, the resulting system is much more of a database than an expert system. However, the basic principles of a frame oriented knowledge base still remain. Additionally, the system was built to be as flexible as possible to allow easy modification. Expert system capabilities could still be added at a later date.

3.3 The Basic Firesys Structure

The Firesys system is made up of five primary components. The largest component is the knowledge base. As the knowledge base is currently structured, the data frames are organized into a hierarchical tree, and contain no procedural knowledge. The knowledge base is not composed solely of data frames. It also contains what we have called system or meta frames. These meta frames contain procedural knowledge needed by the system to access the data frames. This procedural information is not to be confused, however, with procedural knowledge that would be used by the expert system to analyze the data. That kind of knowledge has not as yet been included. The system frame procedural knowledge tells the system how to do things like displaying a data frame of a particular type, adding or deleting information from frames or frames from the knowledge base, and how to search the data frame tree for particular information.

The second system component is the knowledge base interface. These functions provide the only legitimate access to the knowledge base. Users of the knowledge base access data through calls to these interface functions. Functions are divided into two primary groups: those that access data frame information and those that access slot description information. Slot value retrieval is considered to be a data frame access. Utility functions are included that add and delete values from slots, and that add and delete data frames from the knowledge base.

The third major component of the Firesys system is the print-package. The purpose of this component is to provide a uniform grouping of functions that can be used to output information to the display of the program user. They act as the sole means by which system components are allowed to present information to users of the system. Functions include the capability to display menus, screen headings, slot titles, and individual slot values. The functions keep track of screen displays, insuring that headings and values are not split up, menu items are numbered properly, menu choices are selected correctly, and that displays of data larger than one screen-full are handled properly. The centralization of these functions serves to make displays somewhat uniform, and greatly reduces the redundancy of display code.

The last two components are two separate programs that utilize the knowledge base. As noted above, all accesses to the knowledge base are performed through the interface functions and all output through the print-package. These two programs serve two different purposes. The first program, the Query system, was designed to provide naive users with a user friendly interaction interface to the knowledge base. Through menus, it allows the user to traverse the data frame tree, accessing any information needed.

The second program, the Builder system, was designed for use by a more sophisticated knowledge base builder, and acts as the knowledge base editor. This program allows the user to traverse the data frame tree, allowing alteration of values and frames. Unlike the Query system, the Builder is expected to be used by an individual with an intimate knowledge of the structure and function of the knowledge base.

These five components comprise the Firesys program structure at this time. The system was purposely designed in this component fashion to allow easy changes in knowledge base implementation, and easy changes in the programs that access it. Because of the clear and specific interface to the knowledge base, internal structures (implementation) of the knowledge base can be changed without affecting the programs utilizing it, and visa versa. This structure allows a high degree of flexibility, and was instrumental to the implementation conversion performed by the author for this thesis project.

3.4 Frames, Default Reasoning, and Representations[2]

As described earlier, frame based systems usually are structured to create a semantic network. Within this network, frame interconnections represent relationships that frames have with each other. These relationships often represent a hierarchy. For example, the "Fido" frame mentioned earlier in this chapter represents a specific instance of a dog. The 'IS-A' attribute (slot) in the "Fido" frame indicates a relationship that "Fido" has with the frame 'dog'. In this case, it indicates that "Fido" is a dog. That is, "Fido" belongs to the greater class of things called 'dog' (please see figure 1). Likewise, if we were to examine the 'dog' frame, we would find that it too has a slot called IS-A and that its value might be 'mammal'. Now there are many creatures that are mammals that are not dogs (i.e. cats, horses, etc.), and there are many dogs that do not have the name "Fido" (i.e. Bandit, Spike, etc.). But, of the creatures that are mammals, all share some characteristics in common. Similarly, not all dogs look like "Fido", nor do they have that name. However, they all have some 'dog' characteristics in common.

---

[2]The following discussion about frames and default reasoning is based on information the writer has gleaned from coursework in Artificial Intelligence and from the following text and paper:

Charniak,E.,McDermott, D., Introduction to Artificial Intelligence, Addison-Wesely, Reading, Massachusetts, 1985.

Greiner, Russell, "RLL-1: A Representational Language Language", Stanford Heuristic Programming Project, HPP-80-9 (Working Paper), Computer Science Department, Stanford University, Stanford CA, October 1980.

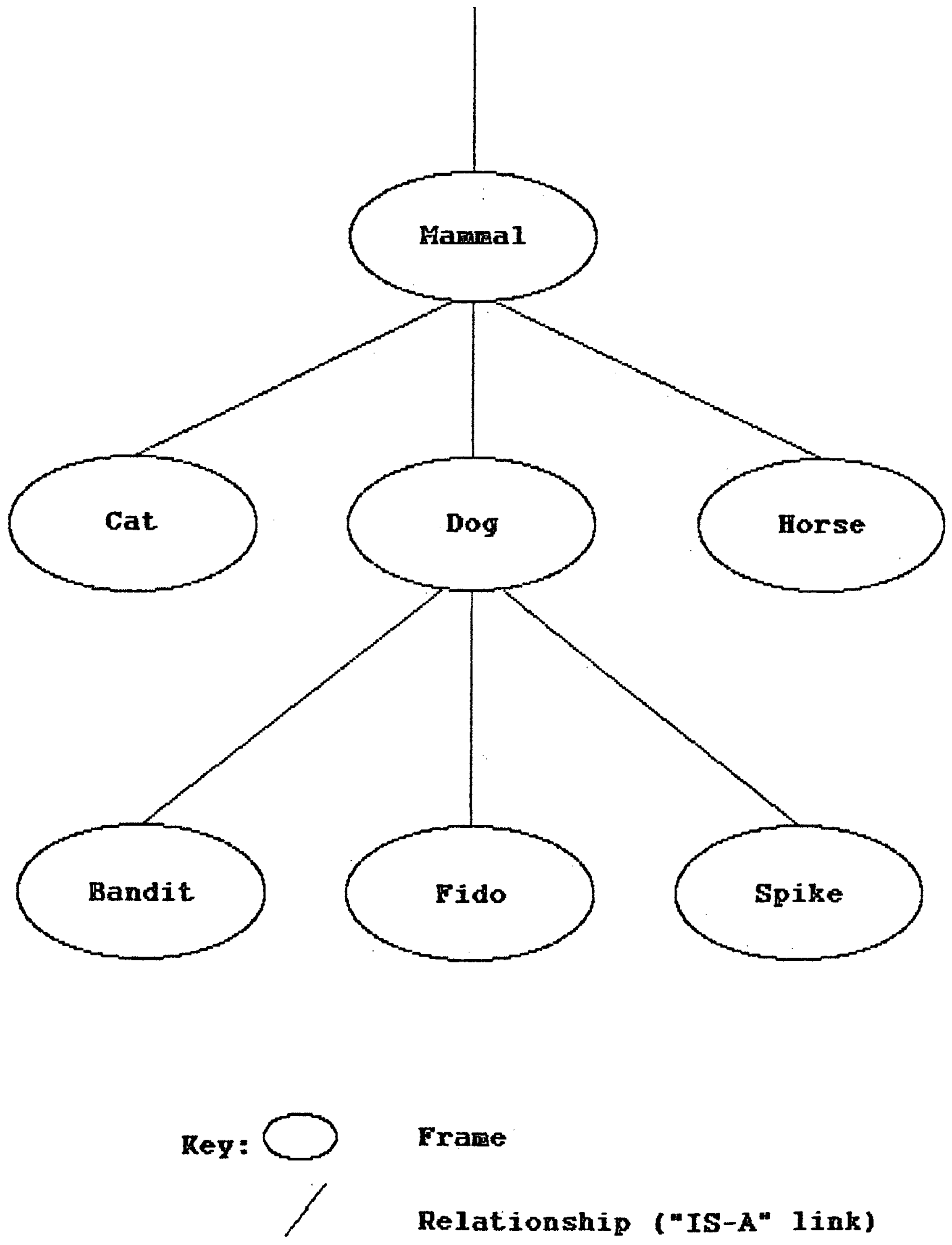


FIGURE 1: Frame Inheritance Hierarchy

These relationships suggest a hierarchy of attributes related to given objects in the world. As one travels up the hierarchy, one finds information that is more general but still common to only the objects below it. Moving up farther, we reach classifications that apply to more and more classifications of objects. Likewise, as we move down the hierarchy, information becomes more specific to narrower classifications of objects. This narrowing continues until we reach individual object instances. At the lowest level we have totally specific information about a particular object, and at the highest, information that applies to all objects.

An important concept associated with knowledge hierarchies is the idea of inheritance. The notion is essentially the idea that objects lower in the hierarchy "inherit" the characteristics of objects that are higher in the hierarchy (from parent nodes). From the "Fido" example, we can see that Fido is a dog because his parent node in the hierarchy ("IS-A" link) is "dog". If we wished to find out about Fido's characteristics, we would first examine the values of attributes local to the "Fido" frame. To find out more about what makes Fido a dog, we would move up to the "dog" frame and examine attributes there. Fido inherits those characteristics. Likewise, one could again move upward from the "dog" frame to the "mammal" frame to inherit more characteristics. In this way, one can obtain a full description of "Fido".

This form of inheritance is also often called default reasoning. This is due to the fact that if the characteristic is not specific to the node we are at, then the value defaults to the characteristic contained in the class to which the node belongs. In this case, the class node is the IS-A linked node. The system reasons that unless otherwise stated, the superior class characteristics apply.

The main idea behind a hierarchy is that specific attributes that belong to individuals are lowest in the hierarchy, while characteristics that are common to wider and wider groupings of individuals are located higher in the hierarchy. This structure allows for drastic reductions in the redundancy that would be present if each individual needed to be described completely.

However, semantic networks are not necessarily trees, although a particular one could be. As the name implies, they are networks. This means that some relationship paths may cycle back to a starting node, allowing an object to circularly define itself. If so, how can there be a hierarchy? Well, the network represents a combination of many hierarchies. If one were to extract only one hierarchy (i.e. biological classification), one would have a taxonomic tree some what similar to that seen in figure 1. This capability to combine many configurations of information relationships is another powerful feature of semantic networks. The Firesys system uses three such hierarchies.

The production of three hierarchies within the Firesys system was primarily a result of the group's exposure to RLL-1 [Greiner,1980]. RLL-1 is a special language used for building knowledge bases at Stanford University. The initials RLL stand for the words

Representation Language Language. It allows its user to develop a representation scheme (language) for frame oriented knowledge bases. It acts as a system building tool that creates a knowledge base environment.

The main power of RLL-1 is that it not only allows one to specify the structure of frames and their relationships, but it also allows one to specify characteristics of the slots contained within the frames. Within RLL-1, slots are categorized into types, and each type is described by another frame. This frame may contain procedural information. Functions that access the slot can use the associated procedures to perform appropriate operations on the slot. This idea of treating slots as basic objects that have their own procedural capability, was directly incorporated into the Firesys system, and forms one of the three hierarchies.

The slot description hierarchy provides information that the Firesys system uses to maintain and manipulate the knowledge base. The hierarchy is therefore part of the system frames and separate from the actual data. In other words, the slot hierarchy contains system procedural knowledge.

In addition to the slot description information, the Firesys system needed to have frame description information. This type of information moves one level higher, describing frame characteristics, and providing procedural information associated with frame manipulations. This information, like the slot description information is grouped into a hierarchy, and forms the second hierarchy of the system. Also like the slot level information, this hierarchy is contained within the system frames, as it too describes knowledge base manipulations.

The third hierarchy present within the Firesys system is contained within the data frames themselves. As noted earlier, this hierarchy contains no procedural knowledge at this time. It only represents a breakdown of a mass of information associated with plant species, ecosystems, and associated fire effects. Each level in the data frame hierarchy essentially provides a more detailed look at information specific to the frame above it.

3.5 Firesys Data Frames

As indicated above the Firesys data frames form a hierarchy that is represented by a tree. The organization of that tree is illustrated in figures 2 and 3. The root of the tree is a permanent frame called "Superior". Currently, all entry to the knowledge base is performed by accessing this frame. It contains pointers to the primary components of the knowledge base structure. This frame serves no purpose other than to bind the portions of the system together and to provide a uniform entry point.

There are two primary information components of the data frame portion of the knowledge base: the ecosystem level information, and the species specific information. The species side of the knowledge base tree

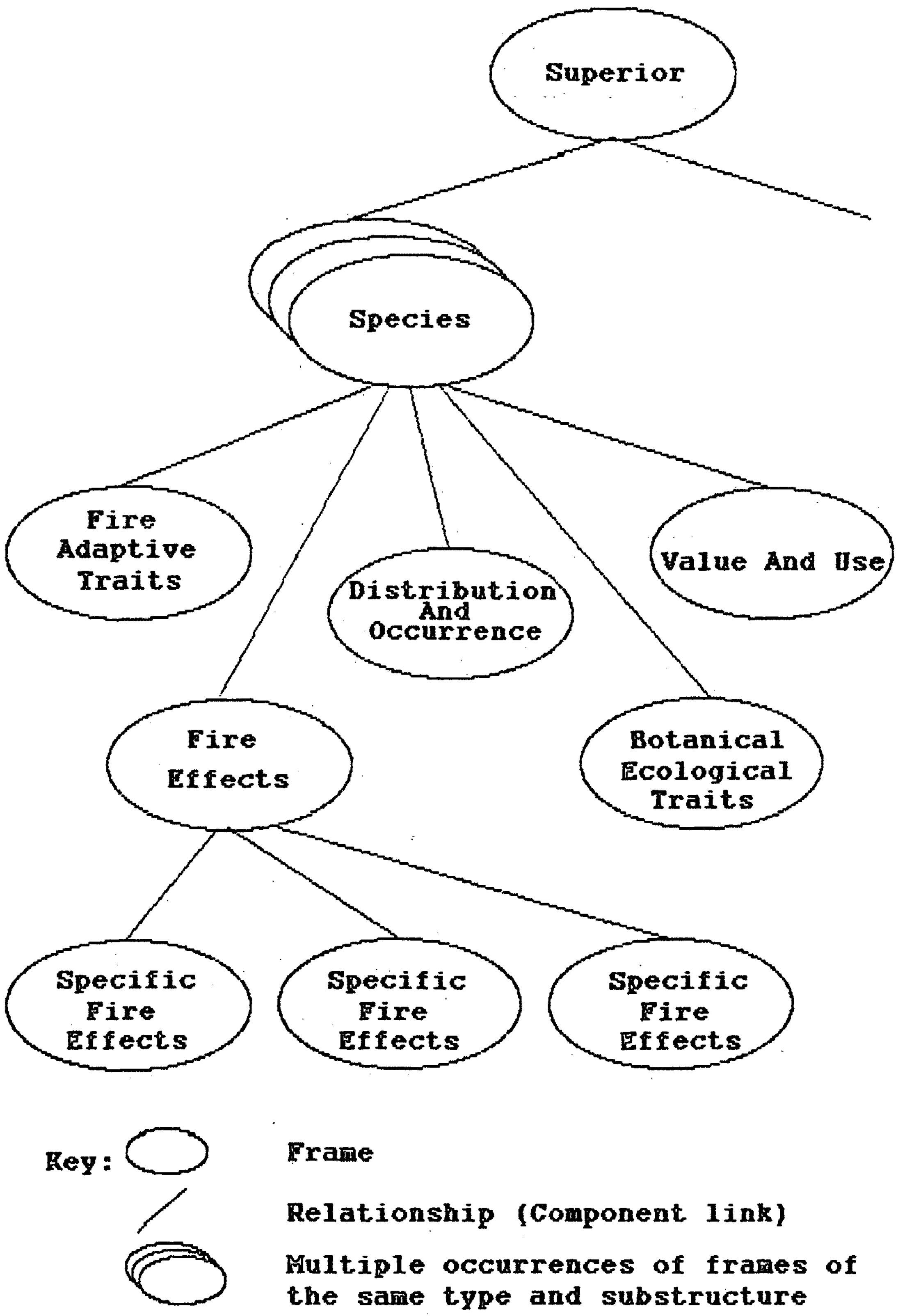

Superior
Species
Fire Adaptive Traits
Distribution And Occurrence
Value And Use
Fire Effects
Botanical Ecological Traits
Specific Fire Effects
Specific Fire Effects
Specific Fire Effects
Key:
Frame
Relationship (Component link)
Multiple occurrences of frames of the same type and substructure

Figure 2: Data Frame Structure of Species side of Knowledge Base

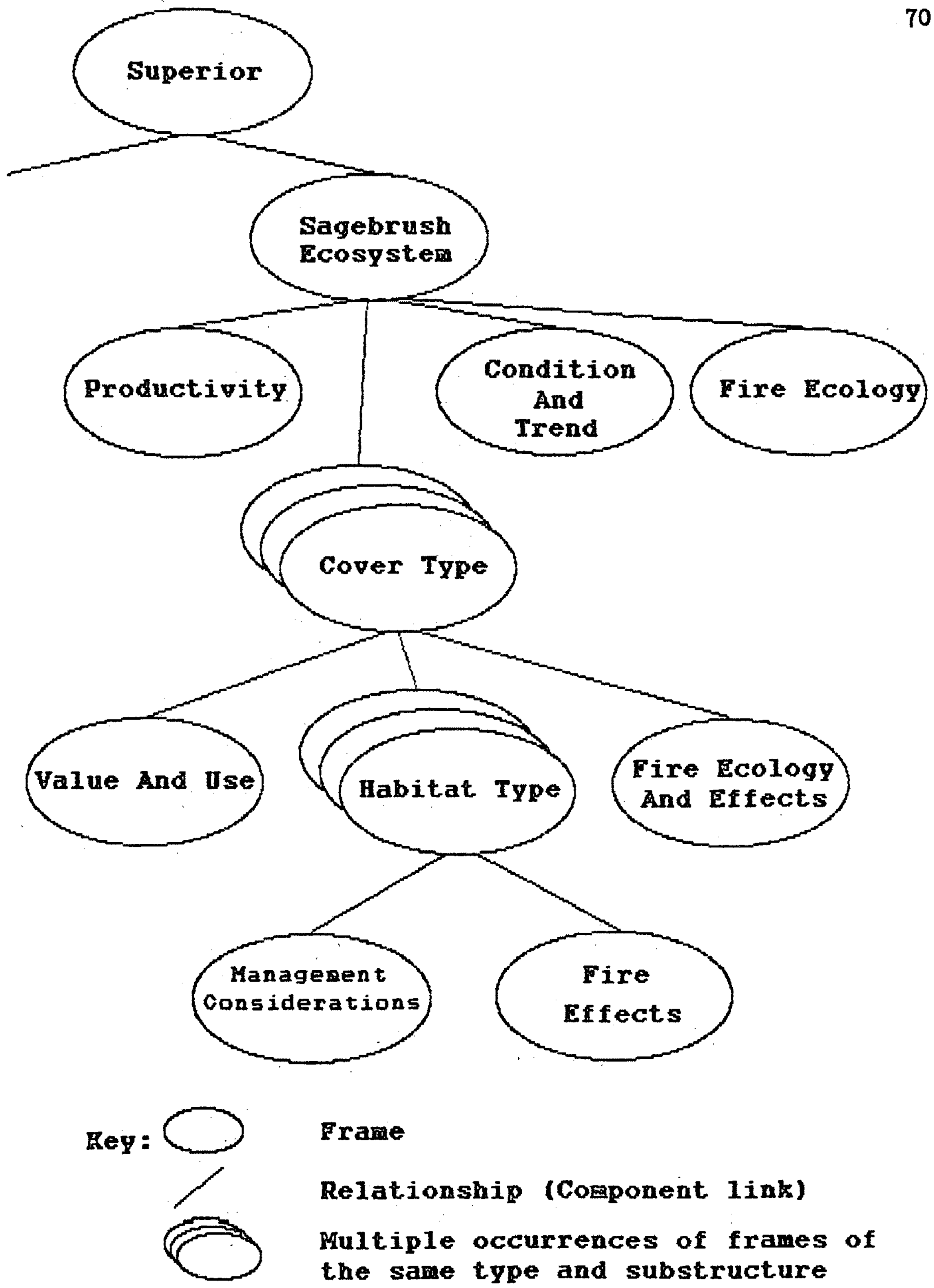


Figure 3: Data Frame Structure of Sagebrush side of Knowledge Base

contains information organized by plant species (please see figure 2). There are multiple instances of species type frames within the knowledge base, and each is directly accessible through the "Superior" frame. Species frames additionally have Subframes, each of which contain more specific information about that species.

A species frame contains the species scientific name, common names, life form, some other general information, and pointers to subframes containing information specific to particular domains. Each species frame has the same type of slot structure and the same type of subframes. Each species frame instance has its own subframe instances associated with it. For example, every species frame has a slot named "Value And Use" which holds the name of the frame containing the information associated with that domain that is specific to that species.

Likewise, a subframe might also have its own subframes. Within the current structure of the species side of the knowledge base, only the "Fire Effects" frame has subframes. The "Fire Effects" frames contain general statements about fire effects specific to the parent species. The "Specific Fire Effects" subframes contain more detailed information that is specific to actual burns of different severity performed at different times of the year.

As one can see, more specific information is stored lower in the tree. This is consistent with the hierarchy description provided earlier, and might lead one to believe that an inheritance hierarchy exists.

However, the inheritance utilized at this time by this side of the knowledge base is minimal. The only inheritance that occurs is associated with the species name that a subframe identifies itself as possessing. All subframes of a species inherit the species scientific name. This name is utilized when the related subframe information is displayed so that a user knows to which species the information is related.

Similarly, the ecosystem side of the knowledge base contains information grouped by level of specificity with regard to ecological groupings of plants (please see figure 2). One enters the sagebrush ecosystem portion of the system by directly accessing it from the "Superior" frame. There is only one sagebrush ecosystem frame. At this level, information that applies to the ecosystem in general can be accessed. More specific information about foliage productivity, condition and trends, and ecosystem level fire ecology can be accessed by moving to one of the immediate subframes. Additionally, the ecosystem can be further broken down into cover types of which it is composed.

Cover types provide yet another level of greater specificity of information. Like species, there are multiple instances of cover types (please see figure 3). The user can choose a cover type from the ecosystem frame, and then access this more detailed information. Again, yet more detailed cover type specific information (Value And Use, and Fire Ecology and Effects) is available in immediate subframes.

Cover type specific information can be subdivided even farther. Under cover type, information has been grouped into habitat type subdivisions. Like moving from the ecosystem frame to the cover type frame, the user can proceed from a specific cover type to a specific habitat type. At this level, habitat specific information is available. Also available, is yet more specific information regarding habitat management and fire effects. This information currently represents the most specific level of information accessible.

An important point that should be stressed here is the flexibility of the system. Over the past year, the Firesys system has under gone many changes. The frame structure utilized has allowed these changes to be performed without excessive effort, and insures that future restructuring and modification is possible. This capability is the real power of this system. When one compares it to standard data bases, one finds this to be the case.

3.6 Firesys System Frames

The key feature of a frame oriented knowledge base is its inheritance capabilities. Although limited within the data frames, the system's use of inheritance is heavily imbedded within the system frames. As mentioned earlier, the system frames are composed of two inheritance hierarchies. One being frame oriented, and the other slot based.

The frame oriented hierarchy provides a means by which information, both descriptive and procedural, about different kinds of frames can be stored in a central location within the knowledge base. As one can see from figures 2 and 3, there are currently eighteen different types of frames. All but five of these frame types have multiple instances. For example, a species type data frame exists for each plant species that was entered into the system. For each of these species data frames, there are five subframes, each of a different type. One of the subframes (Fire Effects) is additionally allowed to have multiple subframes of its own. Therefore, except for the 'Superior', 'Sagebrush Ecosystem', 'Productivity', 'Condition and Trend', and 'Fire Ecology' frames, each frame type has many copies that contain different values and are associated with different super and subframes.

For each of these frame types a frame descriptor was created (called a meta-frame). All information describing a frame of a given type and the procedures used to manipulate that frame are stored within this frame descriptor. In this way, information that is common to frames of one type is stored in one location. The actual frame instances contain only the values that are specific to it, and a value identifying its type.

Access to frame level information is always performed by directly accessing the desired frame instance. For instance, if one wanted to know the value of a species' name, one would request the specified frame to give the caller the value stored in the 'SPECIES' slot:

```
(get-data-frame-slot 'species4 'SPECIES)
```

Such a call would return a value like "Sitanion Hystrix". However, if the information desired was not a value specific to the 'species4' frame, the system will automatically go to the frame descriptor for this type frame to retrieve the needed information. As illustrated in figure 4, a call to retrieve the list of slots that are valid in a species frame would first cause a search of the specific data frame. Not finding the needed value there, the system would automatically search the meta-frame (frame descriptor) associated with the data frame for the value. In this case the needed list is located and returned. If the value is not found in either place, an error message is returned. As can be seen, this hierarchy is only one level deep.

The second hierarchy, the slot oriented one, is similarly structured. In this case, however, the type of information retrieved is primarily procedural in nature. The slot descriptor frames contain information on how to display a slot and its value to the screen, and on how to add and delete values to and from a slot. If one wished to display a slot and its value on the screen, one would retrieve the procedural code stored

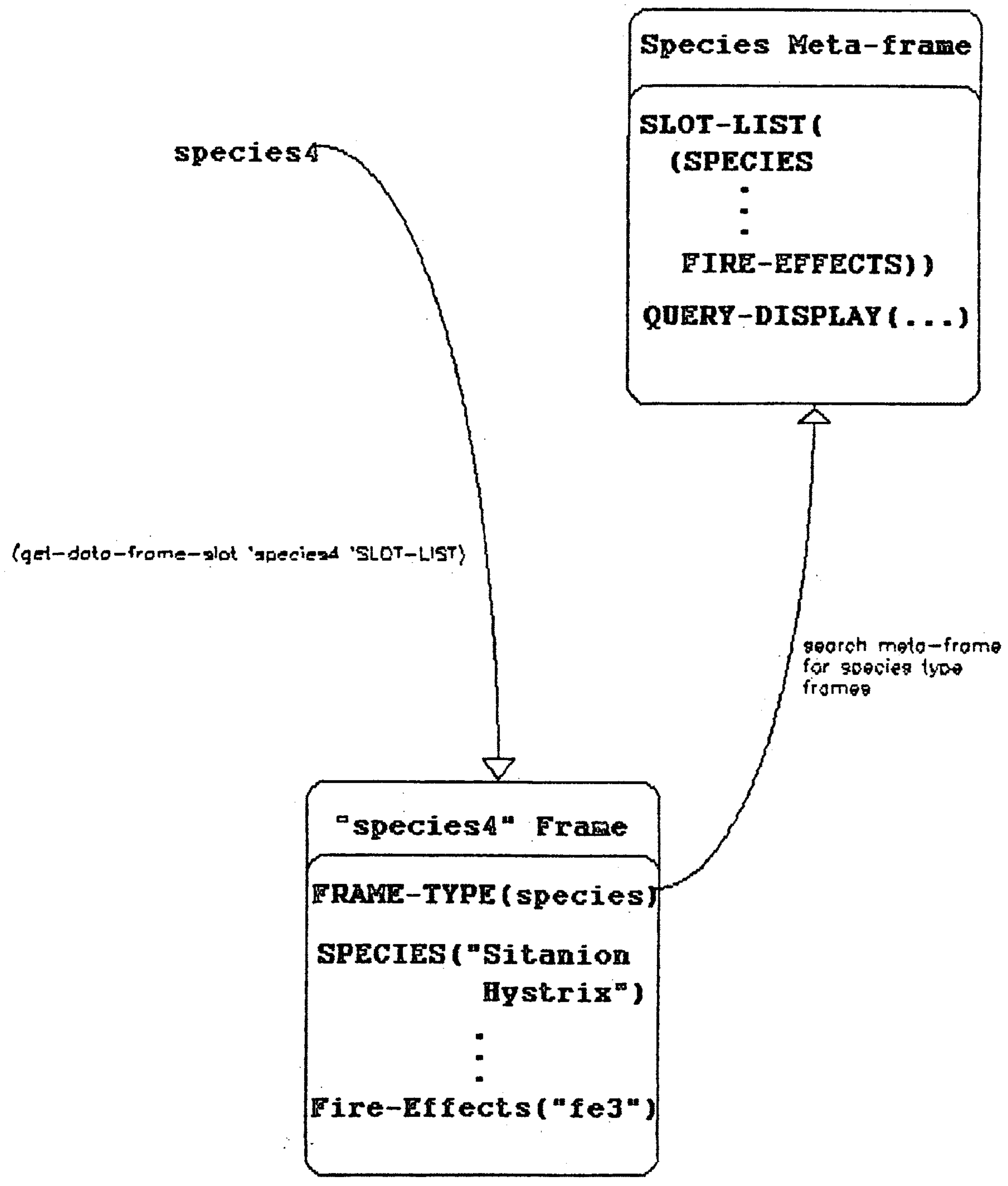


Figure 4: Search sequence performed when slot value is requested and not resident in data frame

(Note: All species frame identifiers are retrieved from a name table contained in the Superior frame) in the slot descriptor frame and apply it to the given data frame. As an illustration take the following function call:

```
(funcall
  (get-slot-descriptor-slot 'SPECIES 'QUERY-DISPLAY)
  current-frame-name)
```

This Lisp function call would cause the code for displaying a slot in a format that the Query portion of the system needs, to be retrieved from the SPECIES slot descriptor frame. It then would execute that code using the current frame identifier. This code knows how to retrieve the data value from the data frame and then how to display it, with a heading and properly formatted.

For each unique slot name in the system there is a corresponding slot descriptor. However, many of the slots hold the same type of information and require the same procedures for manipulation and display. It would be highly redundant to house the same code in each slot descriptor frame. To avoid this redundancy, six groupings of slot types were identified. Slots could be classified according to their contents. Slots were found to contain:

1) single values (atom slots)
2) lists of values (list slots)
3) text (text slots)
4) heading text only (header slots)
5) single frame identifiers (pointer slot)
6) lists of frame identifiers (pointer list slots)

Based on these six classifications, slot class frames were constructed. Like the meta-frames (frame descriptor frames), the slot class frames contain information common to all slot descriptor frames of the same classification.

When making a call to retrieve descriptive and/or procedural information related to a slot, the system follows the same steps as it does with data frames. It first looks for the desired slot and its value in the slot descriptor frame. If the information is not found there, a search is made of the slot class frame. Figure 5 illustrates this process. If one wished to display the 'SPECIES' slot of the 'species4' frame in Query format, the following call would be made:

```
(funcall
  (get-slot-descriptor-slot 'SPECIES 'QUERY-DISPLAY)
  'species4)
```

The get-slot-descriptor-slot portion of the call would first cause the system to examine the Species slot descriptor frame for the Query-Display slot. Not finding the Query-Display slot there, the system would then examine the slot class frame of class 'atom'. Like the data frames, the slot descriptor frames contain a slot identifying their type. In this case, as seen in figure 5, the SPECIES slot is of type 'atom'. A search of the atom slot class frame locates the Query-Display slot, and the code contained there is returned.

The need to apply the code returned to the identifier of the currently accessed frame points out an important difference between the frame oriented hierarchy and the slot oriented one. Within the frame hierarchy, any executable code found is automatically executed. In the

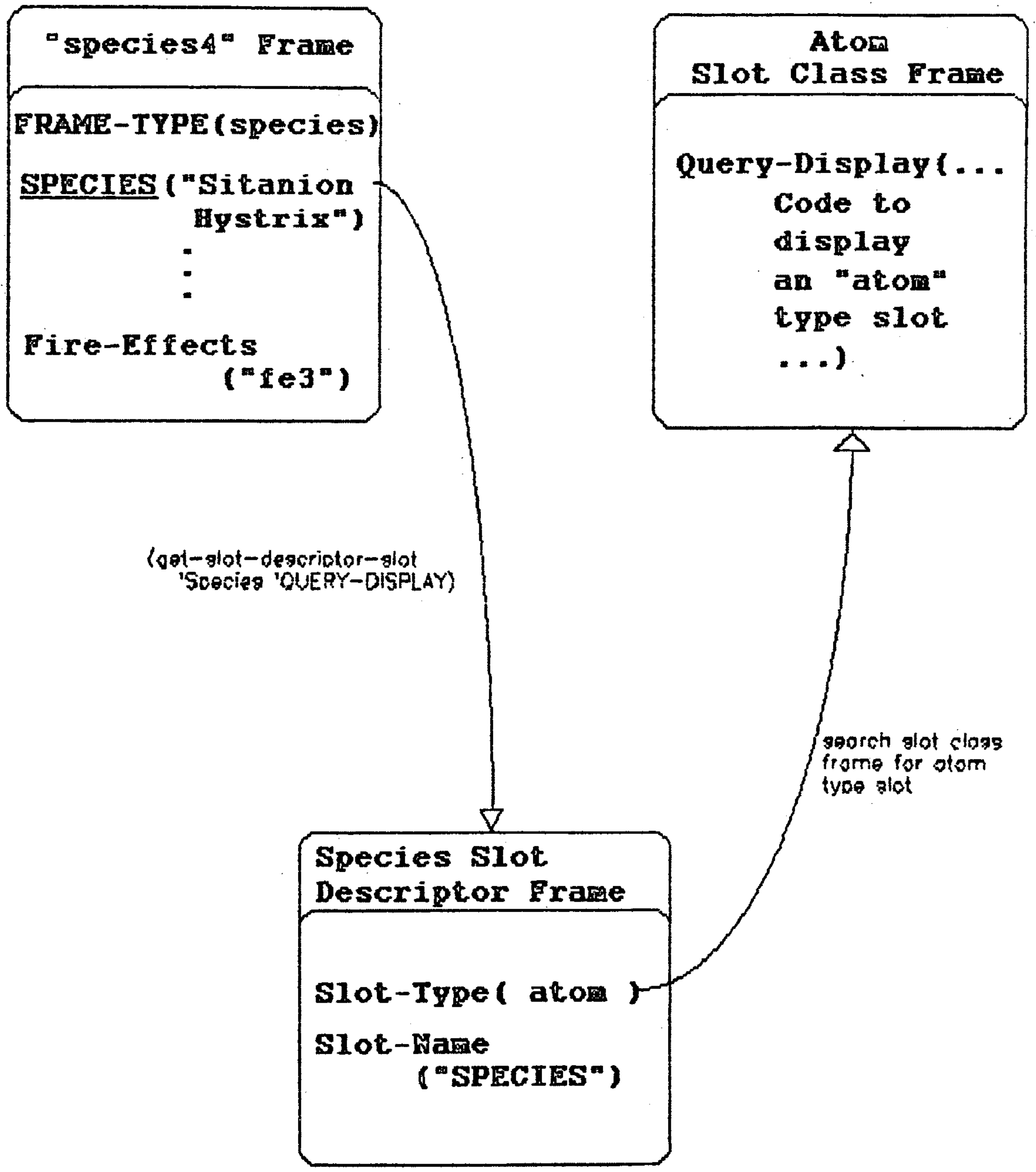

"species4" Frame
FRAME-TYPE(species)
SPECIES("Sitanion Hystrix")
Fire-Effects ("fe3")
Atom Slot Class Frame
Query-Display(... Code to display an "atom" type slot ...)
(get-slot-descriptor-slot 'Species 'QUERY-DISPLAY)
search slot class frame for atom type slot
Species Slot Descriptor Frame
Slot-Type(atom)
Slot-Name ("SPECIES")

Figure 5: Search sequence performed when slot value is requested and not resident in slot descriptor frame

(Note: The name of the current slot being accessed in the data frame is used to retrieve the slot descriptor frame information) slot hierarchy, the caller must explicitly execute the retrieved code. This execution was left to the caller in the case of the slot hierarchy due to the need for extreme flexibility. The kinds of operations performed on slots varied to a much greater extent than did frames, as did the information that might be passed to the retrieved code. However, in the writer's opinion, this flexibility did not prove to be a requirement. The structure of the slot descriptor calls could be made identical to those of the frame descriptors. In any case, except for this difference, the structures are identical.

Going back to the semantic network structure described earlier, one can now perhaps see the usage of default reasoning within this system. The data and slot descriptor frames form the lowest levels in each of their respective hierarchies. Information is initially sought at that level. Having not found any instance-specific information, the system then defaults to utilizing information specific to the class to which the instances belong. In this case, meta-frame or slot class frame information is used. The instance inherits the class characteristics.

3.7 Relationship to Object-oriented Concepts

The central idea of this thesis is that the frame based system which the Firesys team developed is also an object-oriented one. Others have noted that there is a great resemblance between the "LISP-AI" notion of frames and object-orientation [Rentsch,1982]. In this section, similarities will be drawn between object-oriented concepts and frame based representation systems. In particular, similarities between the Firesys system and object-orientation will be shown.

In Chapter 2 of this paper, four main concepts were presented that were associated with object-oriented programming. These concepts were the object, the message passing system, the class system, and the class hierarchy inheritance. All of these components are found within the Firesys system.

An 'object' was defined as an entity containing some private memory and having procedures associated with it [Goldberg,1983]. A crucial property of an object is that its private memory can only be manipulated by its associated procedures [Goldberg,1983]. If one examines the concept of the frame, some similarities to object-oriented concepts are found. A frame is composed of slots. Slots act as the frame's private memory. Slots can contain executable code (procedures) that are specific to manipulations of that frame. These frame features parallel those of the object[3]. However, frames do not strictly enforce these concepts. The stored procedures may not be the only means for manipulation of slot contents (private memory). Slots may be accessed directly, without necessarily using the frame specific procedures. Even so, if the system builders wish, they can incorporate these conventions into a frame based system.

[3]Application of the concept of the 'object' is not only restricted to a frame. System builders can also conceptualize slots as objects in their own right!

Within the Firesys system, some of these conventions were applied. Frames are treated as entities with frame specific internal values and associated manipulation procedures. Although slot contents can be examined without usage of frame or slot specific procedures, alteration of slot values are performed solely by associated procedures. Frame specific procedures for displaying frame contents are also present. Except for the direct access capability, this set-up directly parallels the object description provided above. If the slot accessor functions had been stored in a higher level system frame, then this exception would be eliminated.

Within the Firesys system we went one step farther. Not only are frames treated as objects, but slots are likewise conceptualized as objects. Slots have associated with them procedures and private values. Procedures are associated with slots which provide a means for altering their contents and displaying the slot itself. Additionally, slots have a value for the string to be used when displaying their name as part of the display of the slot. Access to these values and procedures is confined to the same restrictions as the frame accesses.

Another important feature of an object-oriented system that was not mentioned is the idea that objects should act as animate (i.e. active) entities [Rentsch,1982]. This characteristic can easily be incorporated within a frame based system by forcing accessed frame associated procedures to automatically execute. In this way, frame accesses appear to make computations occur as if initiated by the object itself. The frames then appear to be animate.

Within the Firesys system, frame accesses to slots containing procedural information cause immediate computations to occur, without any additional intervention on the part of the caller. This is precisely what makes objects appear animate. Our frames are therefore object-like in their appearance.

This similarity to objects fails with the current structure of the slot hierarchy. Unlike the Firesys frames, accesses to slot associated procedures does not automatically initiate computations. The caller is forced to initiate the computation himself. This leaves an appearance of slots as static entities rather than animate objects.

Again, the primary difference between a frame and an object is dependent upon how strictly certain conventions are followed. Within an object-oriented environment, the concept of the object as an animate entity, packaged with hidden private memory, accessible only through object associated procedures, is strictly enforced. Frame systems provide a high degree of flexibility, and therefore do not strictly adhere to these concepts unless the system builders decide to do so. Within the Firesys system, the structure satisfies some of the standards for an object-oriented environment, but does not fully meet all the characteristics of defining objects. Changes could easily be made to the system to significantly increase its object-oriented character.

The second primary concept of object-orientation is that of a message passing system. This message passing system is essentially the means by which a user interacts with the objects. It is a sort of communication system. Some signal is passed to the object and a message is returned. Within a frame based environment, this would involve the means used to access and execute slot values and procedures. The message passing system would be the functions used to access the frames themselves. Again, the important feature here is the level of animation of the object receiving the sent signal. As mentioned above, to animate the frames, procedural information would need to be immediately executed upon access.

Another important requirement of a message passing system is the need for message passing to be uniform. A frame based system would therefore require a single function call that would cause values to be returned, or frame computations to occur. An example would be a 'send' function:

```
(send <object> <message selector>)
```

where the function would send an identified object a message selector. The message selector would cause a slot access to occur. The slot value found would be returned or executed if it were a procedure. This send function would act as the uniform interface to the frame network, accessing slots and executing any procedural information found. Optionally, the message selector could also contain arguments to be passed on to any procedures found.

The Firesys system attempts to provide these features with its interface functions. The 'get-data-frame-slot' function provides essentially the same capabilities as those of the send function noted above:

```
(get-data-frame-slot <frame-id> <slot-name>)
```

This function also executes any procedures found when it accesses the named slot. However, it does not allow for any passage of arguments to the found procedure. All executed procedures are passed the same argument, the frame-id.

If this were the only function used to access data in the frames, then it could be claimed that the interface was uniform. However, this is not the case within the Firesys system. There is a second function used to access slot specific information, the 'get-slot-descriptor-slot' function. This function has the same format as the 'get-data-frame-slot' function:

```
(get-slot-descriptor-slot <slot-name> <slot-name>)
```

where the first slot-name identifies the slot 'object' (frame) to access and the second slot-name denotes the message selector (slot to access). As noted earlier, this function does not automatically execute found procedures, and therefore falls short of the specification for a send type function.

It would be possible, with little effort, to alter and combine the existing two interface functions to meet the send function requirement. Frames and slots could be treated as independent objects, each capable of receiving a message selector and having their slot stored procedures automatically executed. Optional arguments to message selectors could also be added (this is a standard feature of Common and Franz Lisp). This would make the interface uniform in character, and allow frames and slots to act as animate objects.

The interface additionally includes functions for adding and deleting values from slots, for creating frames, and functions for reading and writing frame structures from and to disk files. Although part of the interface, and dependent upon the implementation of the frame base, these functions really act as utilities for frame and slot manipulation. These utilities are utilized by frame stored procedures that are executed upon access, and are really not part of the message passing system constructed. Within an object-oriented system they would more likely be methods associated with slot and frame type objects.

The last two primary object-oriented concepts are the ideas of a class system, and the usage of a hierarchical inheritance system within it. Described earlier were the frame concepts of semantic networks, hierarchies within semantic networks, and default reasoning as applied to these hierarchies. The concept of a hierarchy of frames is identical to that of an object class system.

Within an object-oriented system, objects are instances of classes, and classes can be instances of other classes. Values and procedures common to objects of the same class are stored within the class descriptor. Elements common to classes of differing types are stored at the higher level class descriptor of which these classes are instances. Likewise, in a frame system the frames lowest in the frame hierarchy are instances of the parent frames above them. The parent frames contain information that is common to its instances. Similarly, information that is common to parent frames is stored at higher levels in the hierarchy of frames. Instances contain information that is specific to themselves, while the frame at the top of the hierarchy contains the most general information related to all the frames of the hierarchy. Some object-oriented systems, like frame based semantic networks, can contain multiple hierarchies.

Default reasoning is another important feature of frame based semantic networks that is also present in object-oriented systems. As described earlier, traits that are common to a grouping of frames are stored in a frame that is higher in the frame hierarchy for those frames. The frames that belong to this grouping inherit the traits stored within this parent frame. Likewise, within an object-oriented environment, values and code that are common to a group of objects are stored within the class that the object is a member of. The objects inherit these values and code from their class. The more general information is just inherit from locations higher in the hierarchy within both systems. The message passing system of an object-oriented environment provides the capability of inheritance. Builders of a frame base system would similarly have to provide this capability in their knowledge base accessing functions. This is of course exactly what is done when default reasoning is implemented.

The Firesys system provides these same concepts within its system frame hierarchies. As described earlier and illustrated in figures 4 and 5, the system frames have an inheritance hierarchy. Data frames form the instances of the frame oriented system. The meta-frames are the classes next higher in the frame hierarchy. Similarly, slots are the instances of the slot hierarchy. The slot descriptor frames form the first level of classes in the slot hierarchy, and the slot class frames the highest level. The interface functions mentioned previously have incorporated into themselves the capability to search upward through these hierarchies for the information requested.

It is hoped that this comparison has shown the reader the great similarity between object-oriented systems, frame based semantic networks, and the Firesys system. The reader should also understand that there is only a similarity and not an identity. Frame based systems are not purely object-oriented, nor is the Firesys system. However, many of the basic concepts of object-orientation are present.

Noted within the preceding text are some changes the writer suggests would make the Firesys system more object-oriented. To these previous changes should be added two more. Within both frame hierarchies no root node in the trees currently exist. At this root it would be expected to find values or procedures that are common to all nodes below it in the hierarchy. To this end, the writer suggests that all the system utilities that are frame oriented be stored and accessed from a new frame that is superior to the meta-frames. Additionally, all utilities that are slot oriented (i.e. the slot oriented interface functions) should be contained within a similar frame that is superior to the slot class frames. The addition of these new highest class frames, and the alteration of the frame/slot accessing interface functions will bring the current Firesys system much closer to being an object-oriented one.

Chapter 4

THE CONVERSION INTO FRANZ LISP FLAVORS

4.1 Conversion Goals

In the previous chapter, a comparison of the current Firesys system structure was made with what would be expect to found in an object-oriented system. In this chapter, a description will be provided of the attempt made by the author to convert the Firesys system into an existing object-oriented environment. As reported earlier, the Firesys code is written in Franz Lisp. The latest version of Franz Lisp has included in it an object-oriented environment called Flavors. Flavors provides the tools need to fully implement object-oriented concepts. The attempted conversion produced a transformation of the existing custom data structures and data maintenance routines that make up a portion of the Firesys system into the Flavors syntax.

The comparison provided in Chapter 3 suggested that the current Firesys software is not fully in a form that could be called object-oriented. A number of changes in the Firesys system structure were recommended. This state of affairs points to two possible approaches to implementing the conversion. The conversion could involve a direct mirroring of the current Firesys system structure. If the Firesys system is object-oriented in character, then such a mirroring of structure should prove simple to implement. The second approach would be to restructure the Firesys system to make it more object-oriented, incorporating changes suggested in Chapter 3. This might not be as easy as straight mirroring of the current structure, but might have the additional benefit of producing some new configurations that could prove to be useful additions to the Firesys system.

The approach taken was to do both. Initially, the first question to be addressed was whether the conversion into Flavors was at all possible. Direct mirroring of the Firesys structure in Flavors could answer this question. The question as to whether changes could be made to the existing structure to make it more object-oriented could be answered by later modification to the initial Flavors implementation.

There were three changes that the author decided to make to the developed Flavors implementation. First, as noted in Chapter 3, slots within the Firesys frames were conceptually being treated as objects, but actually treated as static entities. Unlike data frames, slot values were not created and manipulated as individuals. Slot values were just part of a data frame. Even so, slots did have a class hierarchy structure, with manipulation information stored in slot and slot-class descriptor frames. This separation of slot values from the slot object structure results in an incomplete object-oriented character. Slot values should be part of the local and private instance variables that belong to individual objects. One change to the structure to be made would be the conversion of slots to full object status by giving ownership of slot values to the slot objects.

The second change relates to the lack of uniformity in the object access functions. As mentioned in Chapter 3, there are separate functions to access data frames and slots. The data frame accessing function (get-data-frame-slot) has the built in capability to search the frame oriented hierarchy for needed information and procedures. It also will automatically execute procedural code found. Likewise, the slot oriented access function (get-data-frame-slot) will search the slot oriented hierarchy for needed information. However, it does not execute found procedural code. The caller must evaluate the returned code if appropriate. This condition seems to have resulted from the incomplete treatment that slots receive within the current Firesys structure. Elimination of the necessity for two different functions for object access could be accomplished when the slots are actually treated as full objects. This elimination of the slot specific access function will result in a uniform communication (calling) protocol.

An important point here is the fact that Franz Lisp Flavors, being an object-oriented programming environment, provides the needed message passing function. It goes by the name of 'send' and has the characteristics of the send function described in Chapter 3. Therefore, usage of the Flavors environment will solve the problem of a lack of uniformity in the calling protocol found within the Firesys system.

The last change that the author wished to incorporate had to do with the utility functions. The comparison performed in Chapter 3 mentioned the fact that there are functions that act as utilities for frames and slots that reside outside the frame structure. Referring to the description of what is to reside in the highest frames or classes of a system, it can be noted that code and data that is most general and applicable to a large group of subframes (subclasses and instances) is to be placed there. By definition the frame utilities are general to all data frames, as are the slot manipulation utilities. These utilities should then reside in a new frame (superclass) within each hierarchy. A 'master' meta-frame should contain the frame utilities, and a super slot frame (superclass of the slot-class frames) should be created. This addition will be the last one proposed.

4.2 Limitations on the Conversion Implementation

This conversion is at heart simply an academic exercise to examine a hypothesis and to investigate the plausibility of making object-oriented modifications to the existing Firesys system. Therefore, it is not a necessity that all portions of the system be converted and/or altered. The main issue at hand is whether the structure of the knowledge base is actually object-oriented and if its implementation can be converted into that of the Franz Lisp Flavors environment. This hypothesis suggests that any effort at conversion should then be centered upon the knowledge base and its accessing functions. Any changes in implementation should be totally transparent to programs external to the knowledge base that are accessing it (i.e. the query and knowledge base editor programs).

The author has been intimately involved with three particular portions of the Firesys project. Specifically, the design of the knowledge base, the design and implementation of the knowledge base interface functions, and the design and implementation of the query system. Although the author did implement the slot accessing utilities, he has not been involved in the construction of any programs that utilize functions that alter slot contents. Specifically, he has not done any work on the knowledge base editor program. Because of this lack of experience, it seemed appropriate that the author only perform the conversion and make changes to those parts of the knowledge base and interface functions that were directly related to the query portion of the system.

These restrictions result in the conversion being limited in scope. The conversion will include the transformation of the knowledge base into objects, with frames being unitary objects composed of slot objects. Additionally, meta-frames will be converted into frame class descriptors with a hierarchy. Slot descriptor and slot-class frames will likewise be converted into a hierarchy of slot object classes. Code stored within these classes will only relate to the displaying of these system objects (query portion). Any code that involves the manipulation of slots (addition and deletion of values) and code that relates to removal and addition of frames will be excluded.

In addition to the the above restriction, the author has included two more. Figures 2 and 3 presented in Chapter 3 illustrated that the data-frame portion of the Firesys system is composed of two primary components: the species related frames and the sagebrush ecosystem frames. With regard to the system frames (meta-frames and slot descriptor frames), both components have very similar structures.

However, the species portion of the system has received the most attention, and has the most understood and currently stable structure. Additionally, this side of the system has the most data inserted into its structure. All levels of the data hierarchy have frames in existence. This situation does not exist in the Sagebrush Ecosystem side of the system. Therefore, conversion will also be restricted to code and data that relates to the species side of the knowledge base.

As the conversion progressed, it became evident that only one knowledge base accessing function would be needed. The Franz Lisp Flavors send function would work appropriately for all object accesses. However, the conversion was to be restricted to the knowledge base. The query program was to experience no changes in its interface to the knowledge base. In order to accomplish this transparency, the get-data-frame-slot function was to remain the same, performing the same actions. This required that the get-data-frame-slot function be recoded using the Flavors send function. Additionally, it required that there be no addition of parameter passing. The conversion, therefore, did not include the addition of parameter passage to the procedural code found when knowledge base accesses are performed.

4.3 The Conversion to Franz Lisp Flavors[1]

The conversion process was approached as one of iterative enhancement. A series of small conversions were attempted first. As each conversion was accomplished and tested, conversion of a new portion of the system was attempted. This process was repeated until all the proposed conversions were completed.

The first portion of the system to be converted was the frame oriented part of the knowledge base. This involved the conversion of existing data frames and meta-frames (data frame oriented system frames) into flavors objects. Conversion of slots into objects was reserved for later conversion. The new frame objects would utilize the existing slot descriptor hierarchy.

The conversion process involved making data frames into Flavors objects. Like most object-oriented environments, Flavors makes individual objects instances of object classes. A class descriptor must first be created from which these object instances can created. Within Flavors, a flavor is the class descriptor. The defflavor function is utilized to create a

[1]The Franz Lisp Flavors code for the conversion can be found in the appendix of this paper.

flavor definition (Appendix A contains all the Franz Lisp Flavors code written to perform the conversion):

```
      (defflavor species (FRAME-TYPE
                          SPECIES
                                .
                                .
                                .

                          SUPERIOR-PARENT)
                        ()
                        :gettable-instance-variables
                        :settable-instance-variables)
```

This definition states that a flavor (frame descriptor) named 'species' is to be defined. It indicates that objects of this flavor will have instance variables FRAME-TYPE, SPECIES, ..., SUPERIOR-PARENT, no mix-in (mix-in's will be described later), and that the values of these instance variables can be retrieved and set by specific calls to their names (messages sent to an instance of the 'species' flavor with the instance variable name as the message selector).

An instance of this flavor is created by applying the 'make-instance' function to the flavor 'species'.

```
      (setq species4 (make-instance 'species))
```

This Lisp expression sets the value of the Lisp object (a global variable) 'species4' to one that identifies an instance of the flavor. For each species data frame, an instance of the species flavors was created. To set a value, say the SPECIES slot value, a message is sent to an instance of the species flavor to set its instance variable to the appropriate value:

```
(send species4 :set-SPECIES "Sitanion Hysterix")
```

This communication expression will set the value of the SPECIES instance variable in 'species4' to the value "Sitanion Hysterix". To retrieve the value stored in the SPECIES instance variable one would use:

```
(send species4 :SPECIES)
```

This message call would return the value "Sitanion Hysterix". A special function was written that performs the species object creation and this value setting process for each species data frame that exists in the knowledge base. This function served the purpose of converting the current data structures into the flavors data structures.

Values stored within the instance variables are to be instance-specific values. Any procedural code that is shared by instances of 'species' objects is to be stored at the 'species' flavor (class descriptor) level. This storage is performed by defining a 'method' that applies to all 'species' objects:

```
(defmethod (species :SLOT-LIST) ()
      '(FRAME-TYPE SPECIES ... SUPERIOR-PARENT))
```

This Lisp expression causes a procedure definition by the name of ':SLOT-LIST' to be associated with the flavor 'species'. When called, it will return a list containing the above indicated values. A method was defined for each each procedural value that was originally stored within the meta-frames of the original Firesys system. This included functions utilized to display the contents of the frame by the query program.

The definition of the 'species' flavor and associated methods, and the creation of 'species' instances was part of the first step in converting the Firesys system into the object-oriented Flavors environment. The remaining species related data frames also needed to be converted. Like the process performed on the 'species' frames, a flavor was defined for each subframe of the species level frame, appropriate instances created. Any associated methods for each were also defined. Once this conversion was accomplished the existing frame format data frames were removed from the system. All species related data frames were then coded as flavors objects.

In order for the conversion to this point to appear transparent to the query program, the 'get-data-frame-slot' interface function had to remain the same with regard to its behavior. The data frames were now Flavors objects and only accessible through the use of the 'send' function provided by the Flavors environment. The 'get-data-frame-slot' function needed to be recoded. This code revision was performed. It involved two changes to the send function. To retrieve a value 'get-data-frame-slot' used the identifier of a frame (i.e. species4) to access the related frame. It did not care about the value of the identifier. On the other hand, the send function needed to know the Flavors-generated identifier of a specific object. This value was stored as the value of the original frame identifier (i.e. the value of species4). The new 'get-data-frame-slot' function would have to take this indirection into account. This required that the identifier be evaluated before it was used with the send function. Looking back, it might have been better to scrap the usage of the original frame identifiers. However, since the conversion was performed incrementally and experimentally, there appeared to be no other choice. If a total conversion were to be performed in the future, usage of the Flavors-generated object identifiers would be highly recommended.

The new function also needed to deal with the case where no value had as yet been defined for a slot (instance variable). In this case, the previous definition of the 'get-data-frame-slot' function caused the value 'no-entry' to be returned. The send function would return 'nil'. A simple check for this condition was also added.
With the conversion of the data frames and the revision of the 'get-data-frame-slot' function, the system could now be tested. It worked flawlessly. As far as the query program was concerned nothing had changed. The new implementation was totally transparent to it. This success set the stage for the next level of conversion.

Slots were still being treated as before. They were essentially static value holders. A hierarchy did exist, however, that held slot specific procedural code. To convert the slots into the Flavors environment would mean the creation of slot objects. For each slot in the species side of the system, a flavor was defined. The flavor definitions needed only contain procedural information; no values were needed to be stored. To be consistent with the previous implementation, however, the TYPE slot was included as an instance variable (even though it served no purpose). Any procedural information that was specific to a slot was coded as a method associated with the flavor of the slot.

There was an important difference between the slot implementation within the original Firesys system and the new Flavors implementation. Slots did not exist as objects (actual instances of frames) in the original implementation. They were really virtual objects. In the Flavors environment, access to methods can only be performed by sending a message to an object, a flavor instance. This fact required that slot objects exist. Virtual slot objects could not be used. One dummy slot instance was therefore created to allow access to the slot flavor methods. This modification still did not address the issue of the separation of the slot value from its slot object. A further modification which does answer this problem is discussed later.

An important difference also existed between the structure of the frame oriented system frames and the slot oriented system frames. Data frames really only utilized one level in their hierarchy. When information was not found in a data frame, the information was searched for one level higher in their hierarchy, at the meta-frames. If slots are treated as object instances, one finds that there are two levels in the slot hierarchy. A search is first performed at the slot instance. It then proceeds to the slot descriptor level, and finally to the slot class level. This hierarchy needed to be reflected in the flavors structure.

The first level is easy, just create slot flavors that correspond to slot descriptor frames. But how does one implement the next higher slot class level structures? This is where the concept of mix-in's applies. A mix-in is a flavor definition that another flavor definition can include as part of itself. All characteristics of the mix-in flavor are included as secondary characteristics of the currently being defined flavor. For example:

```
(defflavor SPECIES ((type 'atom)) (atom)
                   :gettable-instance-variables
                   :inittable-instance-variables)
```

In this flavor definition one instance variable named 'type' is defined which has its value initialized to 'atom'. Note that the mix-in field has the value 'atom'. This indicates that all instances of SPECIES inherit the instance variables and methods of the flavor atom. Methods are first searched for at the SPECIES flavor level first. If the named method is not found, the search proceeds to the first mix-in flavor, namely the atom flavor in this case. The mix-in field might also contain other flavor names, allowing multiple hierarchies to be associated with the SPECIES flavor, but this feature is not applicable to the slot hierarchy at this time.

The atom flavor definition needs no instance variables, and has no mix-in's. It looks like this:

```
(defflavor atom () () )
```

This seems to define nothing. However, it does. Although there are no variables, the definition does allow methods (procedures) to be associated with the atom flavor. These procedures can then be utilized by instances of flavors that use 'atom' as a mix-in flavor. This structure allows the slot hierarchy to be constructed just as it was in the original Firesys structure, within the new Flavors structure.

This arrangement was implemented by creating flavor definitions for each slot descriptor frame in the original system. Flavors were also defined for each slot class frame. Where the slot type was an atom, that slot class flavor was added as a mix-in to the applicable slot descriptor flavor definition. The same was done for all slot descriptor flavors, but adding the mix-in of their correct type (i.e. list, text, etc.).

Methods were defined for all slot class flavors that defined procedures for the display of slots of the given type. An example is the procedure for displaying a slot name and value of type atom:

```
      (defmethod (atom :display) (value)
        (let ((display-list
                  (cons (send self :name)
                        (cons ": "
                              (cons value
                                    (list 'NL 'NL))))))
            (print-slot display-list 'atom)))
```

This method definition allows the caller to send a message to the instance of the slot that is an atom (i.e. SPECIES1) to display itself.

```
      (send SPECIES1 :display "Sitanion Hysterix")
```

The method above defines a list of items that is needed by the print-package to print a slot and its value to the screen (display-list). This list is then passed as a parameter to the called function 'print-slot'. The print-slot function is then executed, displaying the slot.

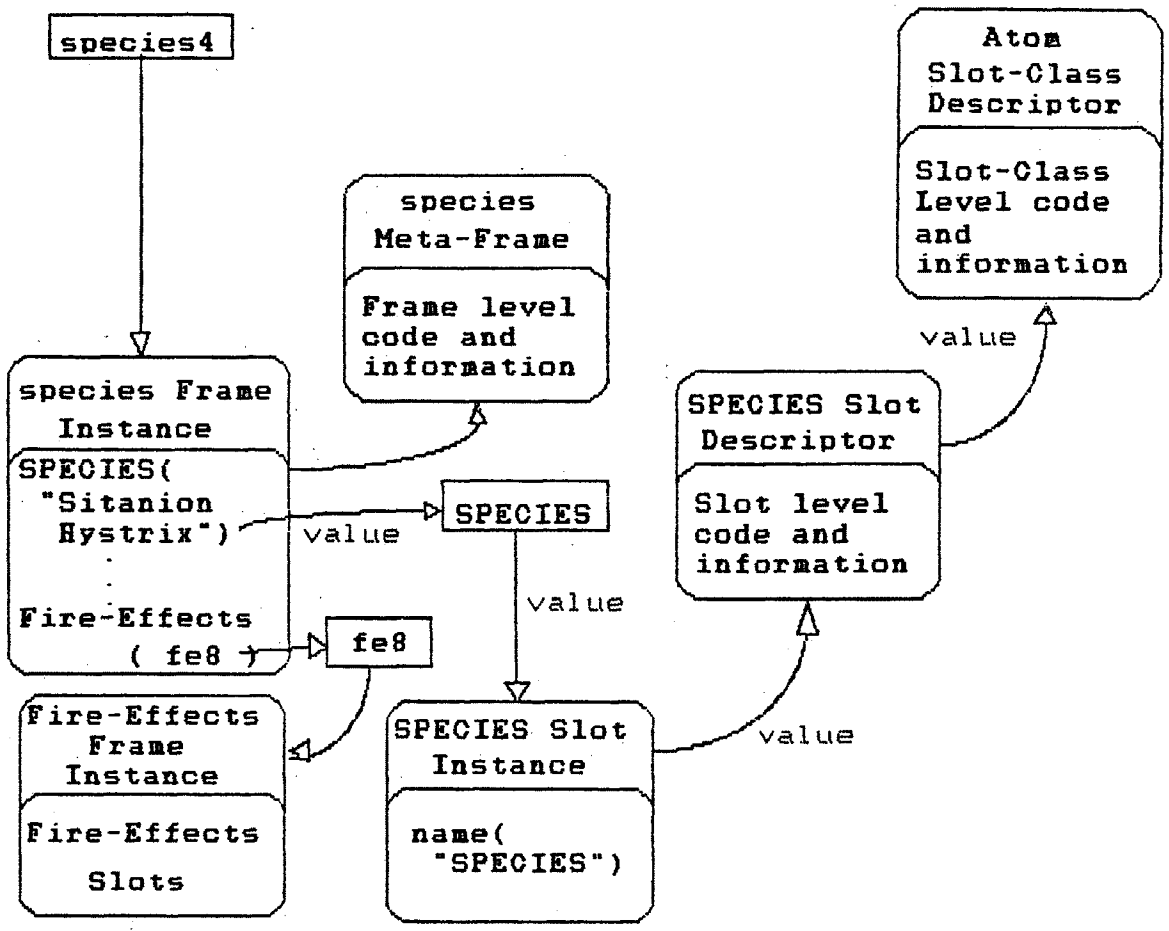
The reader should take note of the two important features of the method definition for ':display'. There is a parameter named 'value' being passed to the method. This passage of parameters directly parallels what the function 'get-slot-descriptor-slot' did in the original implementation. 'value' contains the value found in the slot of the instance variable (i.e. SPECIES slot) in the data frame, and it is the responsibility of the caller to first retrieve and then pass this value to the method. Within the existing system all calls to the 'get-slot-descriptor-slot' function for the display of information (query program) were made from within the procedural code for displaying a frame. This code was housed in the meta-frames for the respective data frames. These calls were easily replaced by a 'send' function call, and being internal to the knowledge base, were totally transparent to the query program.

The second item to take note of is the usage of a variable named 'self'. An interesting feature of the Flavors environment is its usage of this variable. Whenever a message is sent to an object instance, its identifier is bound to this variable. This allows the object's methods to reference other methods associated with itself. In the case of its usage above, it allows the atom method to retrieve the being accessed slot's print name string from the slot's flavor (slot descriptor) one level below where the method is defined in the slot hierarchy.

This also points out an important side effect of this conversion. Within the original Firesys system, when a slot name was printed, the actual slot identifier was used. Under Flavors, this usage of the identifier was too difficult. The author was forced to create a new instance variable within the slot descriptor flavors that contained the string to be used. This creation of a new slot proved to be a solution to problems experienced with the original method. The usage of the slot identifier had created a high degree of coupling between the identifier used and information printed to the screen. Changes in displayed information (i.e. the slot name) resulted in massive updates of system components, defeating the flexibility claimed by the system. Addition of this print string to the slot descriptors eliminated any need to alter other system code, drastically reducing the aforementioned coupling.

The conversion to this point essentially mirrored the structure of the original system within the Flavors environment. Figure 6 illustrates the system organization. As one can see, there is a direct mapping of the frame structures into the new flavors and flavors instances. The system hierarchy has also been preserved through the usage of flavors definitions and flavor mix-in's. The new implementation within the Flavors environment is totally transparent to external programs. The only differences between the original system and the new implementation is the existence of dummy slot instances, and the usage of a print name string when displaying the slot and its contents. Otherwise, the structures are identical. This would suggest that the basic concepts of frames and frame hierarchies implemented in the Firesys system are highly similar if not identical to that of object-oriented concepts of instances and classes.

However, the usage of Lisp atoms as containers for flavor instance identifiers, and the use of dummy slot instances seems to bypass the main concept of the object. An object should be identified by one name. Its value should be an inherent part of itself. To address these issues the author included some additional modifications.


species4
species Frame Instance
SPECIES("Sitanion Hystrix")
Fire-Effects (fe8)
species Meta-Frame
Frame level code and information
SPECIES
value
fe8
Fire-Effects Frame Instance
Fire-Effects Slots
SPECIES Slot Instance
name("SPECIES")
SPECIES Slot Descriptor
Slot level code and information
Atom Slot-Class Descriptor
Slot-Class Level code and information
value

¤	Represents a Franz-Lisp-Flavors-generated value that identifies an instance of a Flavors object.
species4	Is a Lisp variable that contains a Franz-Lisp-Flavors-generated value that identifies a species data-frame instance.

Figure 6: The original system frame structure as implemented under Franz Lisp Flavors.

Note: In this organization of the system, frames that contain values are instances of flavors. Flavors have been defined for the slots and for data-frames, therefore both data-frames and slots represent objects within this flavors environment. Each object is still referenced by a named Lisp atom who's value is the Flavors-generated identifier. A "print name" slot has also been added to the slot flavors to avoid having to pass the slot name. The slot value must still be passed for display.

If a frame is to be composed of objects (slots) and not static value holders, then the values in its instance variables should not be information values but rather slot object identifiers. Modification of frame instance variables to hold slot object identifiers will allow the elimination of the usage of both Lisp atom identifiers, and the need for dummy slot instances. Instead, frame instance variables will act as pointers to slot instances which will house the actual value. Such a reorganization will result in a system that is much more object like.

This reorganization would require two major alterations of the existing Flavors implementation. First, slot flavors would need to add a 'value' instance variable to their definition. Second, the 'get-data-frame-slot' function would have to be modified to take this new level of indirection into account. Value retrieval would now require that first the frame slot value (instance variable) by sending a message to the data frame, and second, the value returned (being a slot object identifier) would be sent a message to return its value.

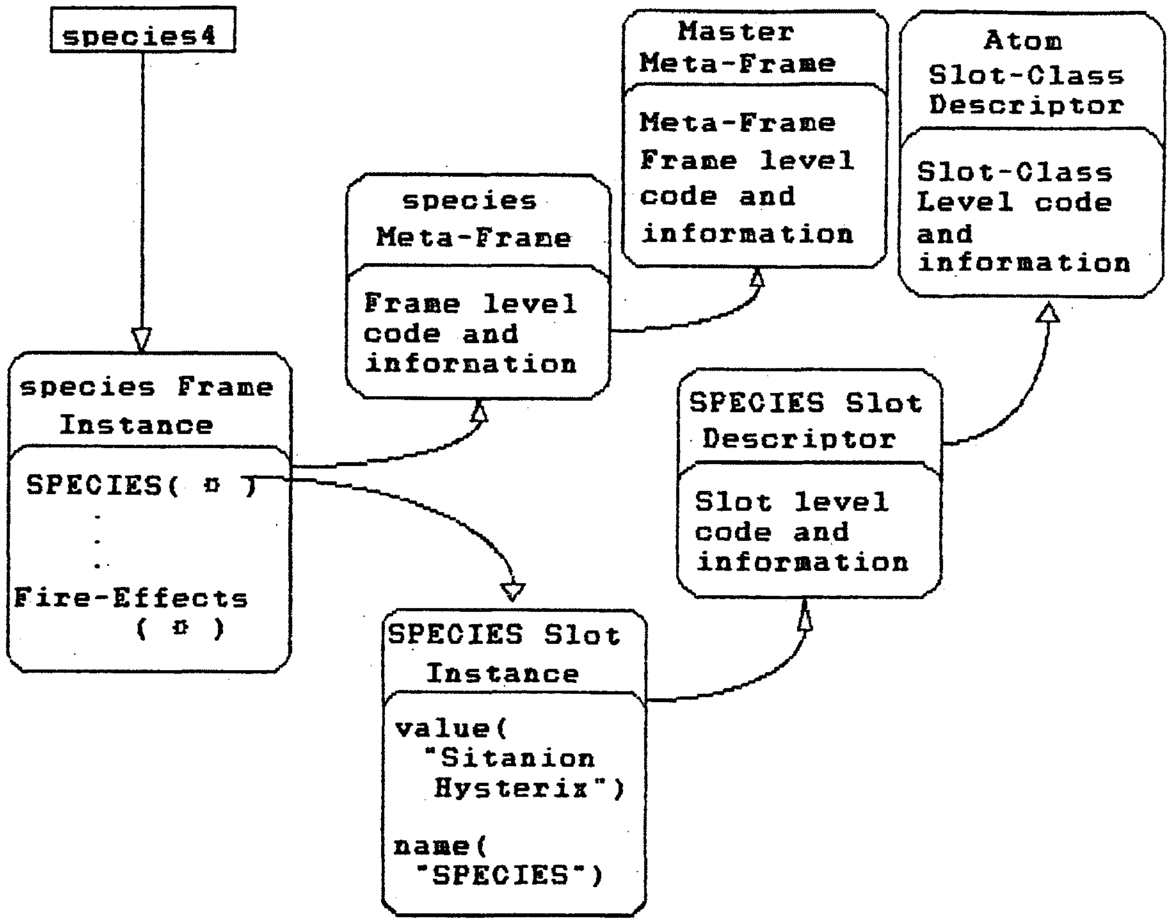
An added side benefit resulted from these modifications. The need for the slot method caller to pass the value of the frame instance variable would no longer be necessary. The slot oriented methods could call 'self' to retrieve the necessary value as needed.

```
(defmethod (atom :display) ()
  (let ((display-list
          (cons (send self :name)
                (cons ": "
                      (cons (send self :value)
                            (list 'NL 'NL))))))
      (print-slot display-list 'atom)))
```

Notice that the new definition of the atom type slot display method no longer needs the passage of any parameters and that the value contained in the slot is retrieved be a simple message sent to 'self'.

There is one more modification that the author included in the final reorganization. As mentioned in the goals and limitations portions of this chapter, utilities that are used by data frames to display their contents should be stored in a new meta-frame that is highest in the frame oriented hierarchy. To meet this goal a new frame oriented master frame was created. Within the Flavors environment, this frame was defined as a new flavor that was 'mixed in' with existing frame flavors. Methods were defined for this new master frame that performed the duties of the utilities. Utility access was performed by meta-frame level methods sending a message to 'self', passing the needed parameters. This alteration served no other purpose than to make the structure seem a little more object-like. Figure 7 illustrates the new reorganized structure. Note that species frames are still accessed via Lisp atom identifiers. This feature could not be changed due to the structure of the query program and the author's lack of knowledge with regard to access code which was designed and implemented by another team member.

Figure 8 illustrates how subframe links should be handled under the new organization. Like frame instance values, the value of slots that are pointers to subframes should be Flavors generated frame object identifiers. Under the author's implementation, these slot values remained Lisp atom identifiers whose values are Flavors generated frame object identifiers.

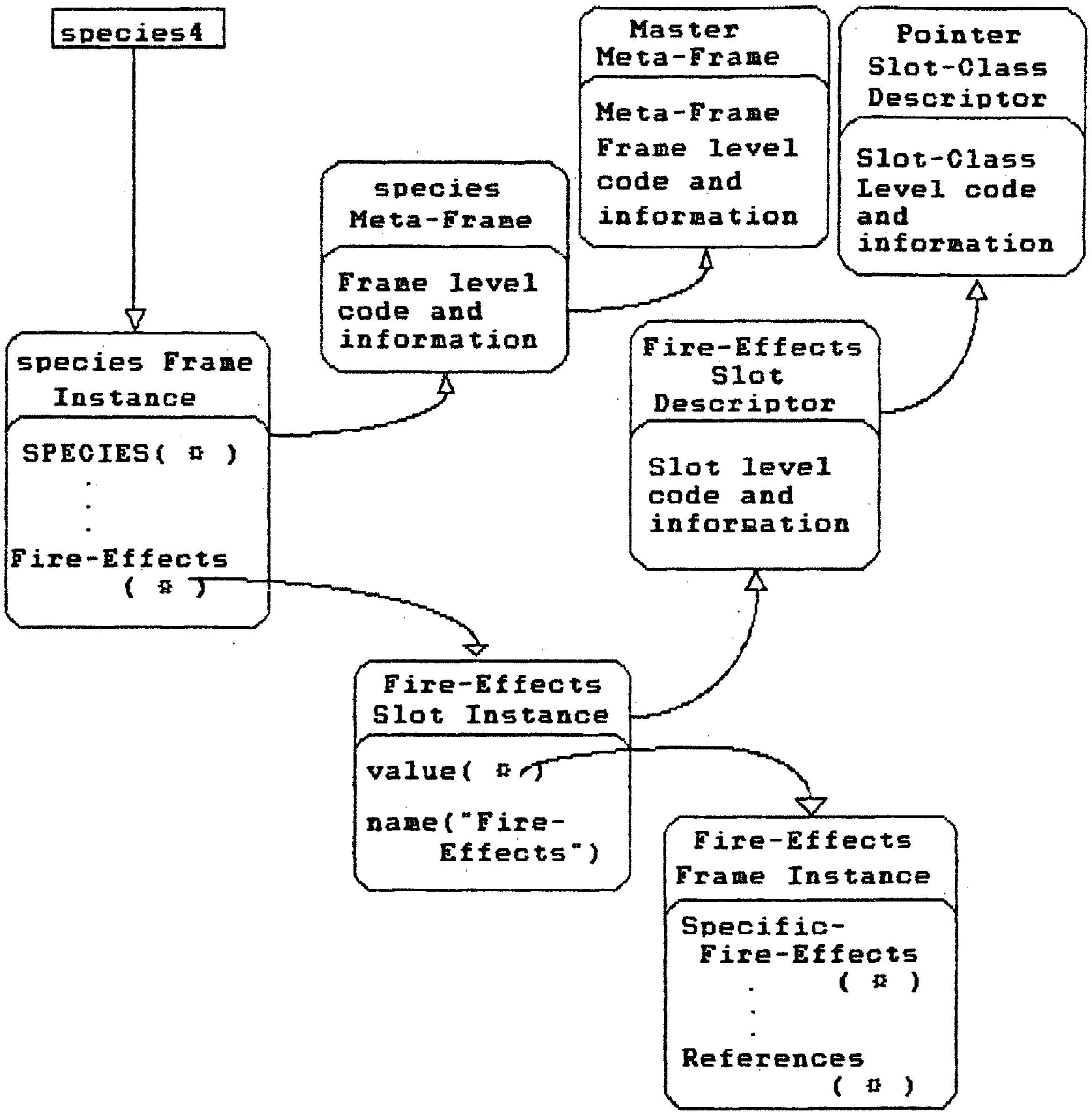

species4
Master Meta-Frame
Meta-Frame Frame level code and information
Atom Slot-Class Descriptor
Slot-Class Level code and information
species Meta-Frame
Frame level code and information
species Frame Instance
SPECIES(¤)
Fire-Effects (¤)
SPECIES Slot Descriptor
Slot level code and information
SPECIES Slot Instance
value("Sitanion Hysterix")
name("SPECIES")

¤ Represents a Franz-Lisp-Flavors-generated value that identifies an instance of a Flavors object.

species4 Is a Lisp variable that contains a Franz-Lisp-Flavors-generated value that identifies a species data-frame instance.

Figure 7: The reorganized system frame structure as implemented under Franz Lisp Flavors.

Note: In this organization of the system, frames that contain values are instances of flavors. Flavors have been defined for the slots and for data-frames, therefore both data-frames and slots represent objects within this flavors environment. Data-frame values are Franz Lisp Flavors values that identify slot objects. Flavors also act as class definitions. Both the slot and the frame flavors have a hierarchy.


species4
Master Meta-Frame
Meta-Frame Frame level code and information
Pointer Slot-Class Descriptor
Slot-Class Level code and information
species Meta-Frame
Frame level code and information
species Frame Instance
SPECIES(¤)
Fire-Effects (¤)
Fire-Effects Slot Descriptor
Slot level code and information
Fire-Effects Slot Instance
value(¤)
name("Fire-Effects")
Fire-Effects Frame Instance
Specific-Fire-Effects (¤)
References (¤)

¤ Represents a Franz-Lisp-Flavors-generated value that identifies an instance of a Flavors object.

species4 Is a Lisp variable that contains a Franz-Lisp-Flavors-generated value that identifies a species data-frame instance.

Figure 8: The reorganized system frame structure with a subframe example.

Note: This diagram illustrates how a subframe is associated with a data-frame instance. The value in the slot instance is a Franz Lisp-Flavors-generated value that identifies the instance of the associated subframe.

4.4 Summary of Results

The attempted conversion demonstrated that the existing Firesys system knowledge base structure could easily be converted into an existing object-oriented environment. What seems most amazing to the author is the ease with which this conversion was accomplished. Having minimal knowledge about Flavors, the author was still able to easily see the parallels between the system frame hierarchy in the existing Firesys system and the flavors concepts. This was a result of the striking similarity between Franz Lisp Flavors' object-oriented concepts and the frame based concepts implemented within the Firesys system. This trial and error conversion process took approximately two weeks of effort. This ease of implementation and the structural correspondence between the original and Flavors' implementation directly support the similarities between frame based systems and object-oriented concepts illustrated in this chapter. It also suggest the high degree of flexibility that the object-oriented approach provides.

An important concept to which this project also lent support was the importance of independence of the knowledge base structure from the external programs that utilize it. The conversion into Franz Lisp Flavors produced a totally new implementation of the knowledge base. The actual data structures and access techniques utilized by the Flavors environment was and is totally unknown to the author. In spite of the drastic change in data structures, the knowledge base still behaved identically with respect to external programs that access it. This independence highlights the importance of defining system components as self contained packages with explicitly defined interfaces. Object-oriented environments support and encourage such an approach. Acceptance of this modularity concept has been demonstrated by this project to greatly increase flexibility.

Modularity is also represented in the class hierarchy constructed, and has resulted in a modification flexibility that would not be seen otherwise. As noted in the preceding sections of this chapter, an incremental approach was utilized in this conversion. The modularity of both the original and the Flavors implementation made this incremental conversion proceed with little or no difficultly. Additions made to the Flavors implementation also proved to be highly flexible and easily accomplished because of this object-oriented modularity. The object-oriented concepts applied within this project have proved to greatly enhance the modifyability and flexibility of the Firesys system.

Chapter 5

DISCUSSION AND CONCLUSION

5.1 Success or Failure of the Conversion

In Chapter 4, a description of the conversion of the existing Firesys system into the Franz Lisp Flavors environment was provided. This description included a statement of goals that were to be achieved by the conversion. In this chapter, how these goals were met by the conversion effort will be examined. Additionally, a discussion will be provided with regard to the pros and cons of utilizing a custom or packaged object-oriented environment. It is hoped that this discussion will address the issue of whether the conversion effort was a success, and whether a packaged object-oriented environment should have been (or should be) used on the Firesys project.

The first goal to be achieved by the conversion was the direct mirroring of Firesys frame structures in the Franz Lisp Flavors environment. The evidence provided in Chapter 4 would suggest that such a mirroring was easily achieved. The primary frame structures of concern were the system frames because of their inheritance hierarchy. If one examines the flavors definitions of the initial conversion and the hierarchy of system frames, one immediately finds a one-to-one mapping of system frames to flavor definitions. Flavors act as descriptors for the objects or subclasses they define, as do the meta-frames, slot descriptor frames, and slot class frames for the data frames and slots of the Firesys system. Each implementation additionally displayed an inheritance hierarchy that behaved identically. These facts strongly support the conclusion that the original system has a very object-oriented character.

There some deficiencies in this object-oriented character however. As noted in Chapter 4, there is an inconsistency with regard to the treatment of object instances within the original Firesys system. Data frames are the main objects of the system. Likewise, data frame objects are the main instances of the Flavors implementation. Here again, one can find a direct mapping between data frame objects in the Firesys system and data frame instances in the Flavors implementation. Where the similarity fails is when one examines how slots are treated in the different systems.

Slots are actually treated identically within both implementations. Each slot is seen as an object. However, within the Firesys system slots are virtual objects. They are not implemented as object data structures. Instead, the slot's name acts as a pointer to a descriptor frame. To implement the original structure within the Flavors environment, the author was forced to create dummy objects to support the object behavior and inheritances characteristics.

Looking back on the Flavors implementation, the author can see an additional way that slot objects could have been implemented. The slot descriptors might have been created as instances of the slot class frames, with the slot names acting as Lisp symbols whose values were the slot instance identifiers. This was exactly what was done with the data frame instances (i.e. 'species4' actually contained the Franz-Lisp-Flavors-generated instance identifier for a species data frame object). This modification would make the implementations much more similar.

This change, however, still does not solve the problem of slots really not being objects. If slots in the original system are objects, then why do they require a separate accessing function? Additionally, why does a user of this access function have to evaluate procedural information found in the slot frame hierarchy? The object-oriented concept of a uniform message passing system is not met, and the basic idea of objects as animate is lost. These two features severely damage the argument that Firesys is object-oriented.

To answer the original question as to whether the Firesys system could be easily converted into an object-oriented environment, one can look at the conversion effort and answer with a resounding "YES". The great similarity between data frames and object instances, between flavors (class descriptors) and system frames, and between the two inheritance systems provides strong support for the notion that frame based systems are object-oriented. The speed and ease with which the conversion was accomplished provides added support. However, the need to treat slots as separate and special objects within the Firesys system detracts from this conclusion.

This leads to the suggestion that parts of the Firesys system might be altered to eliminate these discrepancies. This effort would require that slots be treated as real and not virtual objects, and that the slot accessing function would have to be the same as that used to access any other object (like data frame instances). This could be accomplished by having frame instance variables contain identifiers of slot objects instead of values, and by moving the values into instance variables of slot objects. This is essentially what the author did in the later Flavors implementation, and could easily be accomplished in the current system by adding slot frames. Now instead of conceptualizing frames as being composed of static value holders, they can be made up of slot objects (slot frames) that have their own behavioral and descriptive characteristics. This would add an additional level of indirection, but would increase the flexibility of the system with regard to future enhancements.

Treatment of slots as full fledged objects would eliminate the need for a separate slot accessing function. The message passing (frame accessing) system would then be uniform. Slot procedural information would be automatically executed as it is with frames. Slot object code that requires special arguments would still pose a problem, however. Although the author's experiences with the conversion into Flavors suggests that there are no special arguments, this may not be the case in other parts of the Firesys system. In any case, this problem can easily be addressed by modifying the new accessing function to include optional arguments. The latest versions of Lisp generally include this capacity.

One last addition should be mentioned. The current system utilizes a good number of functions that access and manipulate frames, but that are external to them. In an object-oriented system, by definition, code that manipulates objects must be stored within the class hierarchy to which that object belongs. Within the current system this is not totally true. The system should be modified to house these slot and frame manipulation functions within the respective class hierarchies. This addition would require inclusion of two new frames into the Firesys system frame structure. The two new frames would contain frame and slot utilities respectively, and would act as the root of its hierarchy. All slot frames would inherit code stored in the master slot frame, and all data frames the code stored in the master-frame frame.

These additions to the existing system would make it more object-oriented. They would comprise modifications to the existing Firesys system as implemented in its custom environment. Implementation done within a packaged object-oriented environment such as Franz Lisp Flavors or Smalltalk would also have to take these alterations into consideration.

5.2 Custom versus Packaged Object-Oriented Environment

An interesting question arises now that the conceptual structure of an object-oriented system has been described. Should a packaged object-oriented environment be utilized, or should it be built from scratch? More specifically, should the Firesys system have been built in a packaged environment and should it now be converted? There are two primary factors that influence this decision. First is the question of development time. Second is the question of efficiency and portability.

Building an object-oriented environment can be very time consuming. Many bugs must be worked through, and each "wheel" must be "reinvented" from scratch. A packaged system will already have all the tools needed to implement the object-oriented system. This was exactly the case with the current conversion effort. As a result, implementation is quite rapid. However, the system implementers have no idea as to the composition of the code underlying the packaged system. They must rely on the integrity and efficiency of the packaged environment's functions.

The efficiency issue may be important to a particular application. The choice between a packaged environment and a custom built one is very similar to the choice made by programmers of standard applications with respect to usage of a high-level or assembly language. Packaged environments, like high-level languages, provide many of the tools to build programs quickly and cost effectively. However, their use may lead to a loss in system performance efficiency. Coding in assembly language, although not usually cost effective, may allow the developers to increase system performance to its maximum. Likewise, the choice of building a custom system may result in a more efficient final product.

Within a packaged system little room is left to make modifications to the underlying functions. If how a particular object-oriented environment function interacts with the developed system needs to be altered, it is doubtful that this change could made. The environment's internal code could always be altered, but with little knowledge of its inner workings, this could be disastrous. A custom system allows the developer to "fine tune" the environment to meet the special needs of the developed system. A packaged environment does not.

Beyond the issues of trust in the environment, fine tuning capability, and speed of development, lies the issue of portability. If it is the intent of the developers to produce a system that is not tied to a specific machine, then the issue of portability brings the decision of which form of environment to select to the forefront.

Packaged environments are usually machine specific. This may change in the future, but it seems to be the case now. The Firesys system, from the start of the project, was intended to run on a machine different from that on which it was developed. Development of the system would have been risky if a packaged environment had been utilized. For example, the Franz Lisp Flavors environment could have been utilized. The problem is that none of the other machines on which the project was to be implemented had Franz Lisp Flavors, let alone Franz Lisp. Now, flavors are not specific to Franz Lisp. There are other flavors implementations under different dialects of Lisp. But, examination of these implementations of flavors reveals that there is no standard. Each is different. Another choice would be to implement the project in a language like Smalltalk. It is fairly well standardized, but implementations exist only for specialized machines and micro-computers.

The only choice that is really available to object-oriented system developers who wish to produce a highly portable system is to choose a development language that is standard across the largest number of machines. The choice of usage of a packaged object-oriented environment is really not available in most cases. The Firesys team found that Common Lisp was a language available on most of the target machines that provided the symbolic processing tools needed for development of the Firesys system. On the machines that did not have Common Lisp, it was found that it could be fairly easily emulated. It is within this Common Lisp environment that the current object-oriented/frame-based system was developed.

The developed system proved to be highly portable. When the few system dependent features were extracted from the system, wholesale transfer of the system was accomplished with little effort. These features were essentially restricted to input and output capabilities. Re-coding of these few features produced a system that is essentially identical to the original.

This port[1] demonstrated the importance of system developer's usage of a standard programming environment. If the Firesys system had been originally developed using Franz Lisp Flavors, movement of the system to another machine would have been much more difficult. It would have involved the reimplementation of system manipulation functions that the Flavors environment provides. This is what the custom environment provided in the first place.

One argument can be raised in favor of the packaged environment, however. Usage of a packaged environment leaves the system developers' emphasis on the system to be developed. The presence of object-oriented capabilities help ensure the consistency of the developed system. A custom environment cannot insure this consistency, and may distract developers with environment implementation details. As noted earlier, the Firesys system has some inconsistencies in its treatment of objects.

Once a system is developed and its structure defined, a custom environment can then be constructed. The construction of the custom environment following system development will result in a separation of the developed system from the developed environment and vise versa. The environment builders can then focus on portability and efficiency details without confusing them with structural issues associated with development of the application. This may have been a better approach to have taken with the Firesys system.

[1]The port referenced was to a micro-computer and involved additional alterations to accommodate its memory restrictions.

Individuals developing object-oriented applications will have to wrestle with these development issues. If an application is to be developed for a specific machine, and development takes place on that machine, then the usage of a packaged object-oriented environment seems most appropriate. If the developed product is to be ported to a different machine then usage of a packaged environment depends upon the availability of a portable one. The author would like to stress, however, that usage of a packaged environment may still be very appropriate for applications to be ported to other machines if it is used as an initial development tool. Usage will result in the developed application being more conceptually clean and consistent. A custom environment can then later be added to the application for easy porting.

5.3 Conclusion

This thesis has presented descriptions of a frame based Fire Effects Information system, object-oriented programming concepts, and how the two relate. It was the original hypothesis of the paper that the developed Firesys frame based system was in essence an object-oriented one. The proceeding chapters demonstrated that there is a great similarity between frame based systems utilizing inheritance hierarchies and object-oriented systems. The conversion of the existing Firesys system into a Franz Lisp Flavors implementation strongly supported the hypothesis. Although some discrepancy was found between what one would expect to find within an object-oriented system and the original Firesys implementation, it is felt that the overall structure of the system is inherently object-oriented.

Pursual of this thesis project has also resulted in some recommendations for improvement of the original Firesys system. Upon discovery of some of the improvements, it immediately became evident that the original system should include them, and inclusion has started. Specifically, the addition of the slot "print name" to the slot descriptor frames has proven to greatly reduce some internal coupling that existed in the original, and increase the flexibility of the system. Inclusion of other recommended improvements into the existing Firesys system may also result in system improvements.

It is felt by the author that the thesis project effort has been very successful. It demonstrated the equivalence of object-oriented concepts with frame based constructs in the Firesys system. It also provided a means for examining the Firesys system, and some improvement recommendations. It is hoped that what was learned here will assist the future Firesys developers in their efforts and any other frame based project developers.

APPENDIX

Code Listing of Firesys Conversion to Franz Lisp Flavors

Object-Oriented Environment

FLAVOR AND METHOD DEFINITIONS FOR THE CREATION OF FRAME HIERARCHY SYSTEM FRAMES

```
;************************************************************

;Master FRAME -- frame utilities definitions

;************************************************************

(defflavor frame ()())

(defmethod (frame :query-view-frame-utility)
           (header-fun name-string)
  (send self (find-symbol (string header-fun)
                          *keyword-package*)
        name-string)
  (let* ((slot-list
           (send self :QUERY-DISPLAY-SLOT-LIST))
         (display-list (do ((slot-list slot-list
                                 (cdr slot-list))
                            (displayable-list
                              nil
                              (cond
                                 ((eq (get-data-frame-slot
                                         self
                                         (car slot-list))
                                      'no-entry)
                                  displayable-list)
                                 (t (cons (car slot-list)
                                          displayable-
                                           list)))))
                        ((null slot-list)
                            (reverse displayable-list)))))
        (cond ((null display-list)
               (print-slot
                  '(NL "Sorry no information available on
                    this subject!" NL)
                  "text"))
              (t (do ((display-list display-list
                                    (cdr display-list)))
                     ((null display-list) nil)
                     (send
                        (send self (find-symbol
                                      (string
                                        (car display-list))
                                      *keyword-package*))
                              :display)))))
        (readcontinue))
```

```
(defmethod (frame
            :query-species-print-frame-header-utility)
           (name-string)
  (let ((header (list
           (center-line name-string)
           'NL
           'NL
           (string-append "SPECIES: "
                    (get-data-frame-slot 'self 'SPECIES))
           'NL
           'NL
           HORIZ-BAR
           'NL
           'NL)))
       (print-header header)))

(defmethod (frame :query-print-frame-header-utility)
           (name-string)
  (let ((header (list
           (center-line name-string)
           'NL
           'NL
           HORIZ-BAR
           'NL
           'NL)))
       (print-header header)))


;************************************************************

;SUPERIOR Frame definitions

;************************************************************

(defflavor superior (FRAME-TYPE
               SAGEBRUSH
               SPECIES-LIST
               INTRODUCTION
                    SUPERIOR-PARENT)
            (frame)
           :gettable-instance-variables
           :settable-instance-variables)

(defmethod (superior :SLOT-LIST) ()
   '(FRAME-TYPE
     SAGEBRUSH
     SPECIES-LIST
     INTRODUCTION
     SUPERIOR-PARENT))
```

```
;**********************************************************

;INTRODUCTION Frame definitions

;**********************************************************

(defflavor introduction (FRAME-TYPE
                    SPECIES-INTRODUCTION
                    SAGEBRUSH-INTRODUCTION
                    INTRODUCTION-PARENT)
             (frame)
            :gettable-instance-variables
            :settable-instance-variables)

(defmethod (introduction :SLOT-LIST) ()
   '(FRAME-TYPE
     SPECIES-INTRODUCTION
     SAGEBRUSH-INTRODUCTION
     INTRODUCTION-PARENT))

(defmethod (introduction :QUERY-DISPLAY-SLOT-LIST) ()
  '(SPECIES-INTRODUCTION
    SAGEBRUSH-INTRODUCTION))

(defmethod (introduction :QUERY-DISPLAY) ()
  (send self :query-view-frame-utility
     'query-print-frame-header-utility
     "Welcome to the Fire Effects Information System")
  t)
```

```
;**************************************************************

;SPECIES Frame definitions

;**************************************************************

(defflavor species
     (FRAME-TYPE
      SPECIES
      SCIENTIFIC-ALIAS
      ABBREVIATION
      COMMON-NAMES
      LIFE-FORM
      VARIETIES-AND-FORMS
      DISTRIBUTION-AND-OCCURRENCE
      VALUE-AND-USE
      BOTANICAL-AND-ECOLOGICAL-CHARACTERISTICS
      FIRE-ADAPTIVE-TRAITS-AND-SURVIVAL-STRATEGIES
      FIRE-EFFECTS
      SUPERIOR-PARENT)
      (frame)
      :gettable-instance-variables
      :settable-instance-variables)

(defmethod (species :SLOT-LIST) ()
   '(FRAME-TYPE
     SPECIES
     SCIENTIFIC-ALIAS
     ABBREVIATION
     COMMON-NAMES
     LIFE-FORM
     VARIETIES-AND-FORMS
     DISTRIBUTION-AND-OCCURRENCE
     VALUE-AND-USE
     BOTANICAL-AND-ECOLOGICAL-CHARACTERISTICS
     FIRE-ADAPTIVE-TRAITS-AND-SURVIVAL-STRATEGIES
     FIRE-EFFECTS
     SUPERIOR-PARENT))

(defmethod (species :QUERY-DISPLAY-SLOT-LIST) ()
  '(SCIENTIFIC-ALIAS
     ABBREVIATION
     COMMON-NAMES
     LIFE-FORM
     VARIETIES-AND-FORMS))
```

```
(defmethod (species :QUERY-DISPLAY) ()
  (send self :query-view-frame-utility
     'query-species-print-frame-header-utility
                   "Species Information")
  (detailed-species-info-menu
   (get-data-frame-slot 'self 'SPECIES)))


;************************************************************

;DISTRIBUTION-AND-OCCURRENCE Frame definitions

;************************************************************

(defflavor distribution-and-occurrence
  (FRAME-TYPE
   GENERAL-DISTRIBUTION
   BLM-PHYSIOGRAPHIC-REGIONS
   KUCHLER-PLANT-ASSOCIATIONS
   SAF-COVER-TYPES
   HABITAT-TYPE-INFORMATION
   SPECIES-HABITAT-TYPES
   REFERENCES
   DISTRIBUTION-PARENT)
  (frame)
  :gettable-instance-variables
  :settable-instance-variables)

(defmethod (distribution-and-occurrence :SLOT-LIST) ()
   '(FRAME-TYPE
     GENERAL-DISTRIBUTION
     BLM-PHYSIOGRAPHIC-REGIONS
     KUCHLER-PLANT-ASSOCIATIONS
     SAF-COVER-TYPES
     HABITAT-TYPE-INFORMATION
     SPECIES-HABITAT-TYPES
     REFERENCES
     DISTRIBUTION-PARENT))

(defmethod (distribution-and-occurrence
           :QUERY-DISPLAY-SLOT-LIST)
          ()
   '(GENERAL-DISTRIBUTION
     BLM-PHYSIOGRAPHIC-REGIONS
     KUCHLER-PLANT-ASSOCIATIONS
     SAF-COVER-TYPES
     HABITAT-TYPE-INFORMATION
     SPECIES-HABITAT-TYPES
     REFERENCES))
```

```
(defmethod (distribution-and-occurrence :QUERY-DISPLAY) ()
  (send self :query-view-frame-utility
     'query-species-print-frame-header-utility
     "Distribution and Occurrence Information")
  t)

(defmethod (distribution-and-occurrence :SPECIES) ()
  (send
   (eval (get-data-frame-slot 'self 'DISTRIBUTION-PARENT))
   :SPECIES))

;************************************************************

;VALUE-AND-USE Frame definitions

;************************************************************

(defflavor value-and-use
  (FRAME-TYPE
   DESCRIPTION
   PALATABILITY
   FOOD-VALUE
   COVER-VALUE
   IMPORTANCE-TO-LIVESTOCK-AND-WILDLIFE
   OTHER-USES-AND-VALUES
   ENVIRONMENTAL-CONSIDERATIONS
   REFERENCES
   VALUE-AND-USE-PARENT)
  (frame)
  :gettable-instance-variables
  :settable-instance-variables)

(defmethod (value-and-use :SLOT-LIST) ()
 '(FRAME-TYPE
   DESCRIPTION
   PALATABILITY
   FOOD-VALUE
   COVER-VALUE
   IMPORTANCE-TO-LIVESTOCK-AND-WILDLIFE
   OTHER-USES-AND-VALUES
   ENVIRONMENTAL-CONSIDERATIONS
   REFERENCES
   VALUE-AND-USE-PARENT))
```

```
(defmethod (value-and-use :QUERY-DISPLAY-SLOT-LIST) ()
 '(DESCRIPTION
   PALATABILITY
   FOOD-VALUE
   COVER-VALUE
   IMPORTANCE-TO-LIVESTOCK-AND-WILDLIFE
   OTHER-USES-AND-VALUES
   ENVIRONMENTAL-CONSIDERATIONS
   REFERENCES))

(defmethod (value-and-use :QUERY-DISPLAY) ()
  (send self :query-view-frame-utility
     'query-species-print-frame-header-utility
                  "Value and Use Information")
  t)

(defmethod (value-and-use :SPECIES) ()
  (send
   (eval (get-data-frame-slot 'self 'VALUE-AND-USE-PARENT))
   :SPECIES))

;**********************************************************

;BOTANICAL-AND-ECOLOGICAL-CHARACTERISTICS Frame definitions

;**********************************************************

(defflavor botanical-and-ecological-characteristics
  (FRAME-TYPE
   GENERAL-DESCRIPTION
   GROWTH-FORM
   RAUNKIAER-LIFE-FORM
   GRIME-PLANT-STRATEGY-CLASSIFICATION
   GRIME-REGENERATIVE-STRATEGY-CLASSIFICATION
   REGENERATION-PROCESSES
   SITE-CHARACTERISTICS
   SUCCESSIONAL-STATUS
   SEASONAL-DEVELOPMENT
   REFERENCES
   BOTANICAL-CHARACTERISTICS-PARENT)
  (frame)
  :gettable-instance-variables
  :settable-instance-variables)
```

```
(defmethod (botanical-and-ecological-characteristics
            :SLOT-LIST)
           ()
 '(FRAME-TYPE
   GENERAL-DESCRIPTION
   GROWTH-FORM
   RAUNKIAER-LIFE-FORM
   GRIME-PLANT-STRATEGY-CLASSIFICATION
   GRIME-REGENERATIVE-STRATEGY-CLASSIFICATION
   REGENERATION-PROCESSES
   SITE-CHARACTERISTICS
   SUCCESSIONAL-STATUS
   SEASONAL-DEVELOPMENT
   REFERENCES
   BOTANICAL-CHARACTERISTICS-PARENT))

(defmethod (botanical-and-ecological-characteristics
            :QUERY-DISPLAY-SLOT-LIST)
           ()
 '(GENERAL-DESCRIPTION
   GROWTH-FORM
   RAUNKIAER-LIFE-FORM
   GRIME-PLANT-STRATEGY-CLASSIFICATION
   GRIME-REGENERATIVE-STRATEGY-CLASSIFICATION
   REGENERATION-PROCESSES
   SITE-CHARACTERISTICS
   SUCCESSIONAL-STATUS
   SEASONAL-DEVELOPMENT
   REFERENCES))

(defmethod (botanical-and-ecological-characteristics
            :QUERY-DISPLAY)
           ()
  (send self :query-view-frame-utility
     'query-species-print-frame-header-utility
     "Botanical and Ecological Characteristics Information")
  t)

(defmethod (botanical-and-ecological-characteristics
            :SPECIES)
           ()
  (send
   (eval (get-data-frame-slot
           'self
           'BOTANICAL-CHARACTERISTICS-PARENT))
   :SPECIES))
```

```
;**********************************************************

;FIRE-ADAPTIVE-TRAITS-AND-SURVIVAL-STRATEGIES
;Frame definitions

;**********************************************************

(defflavor fire-adaptive-traits-and-survival-strategies
  (FRAME-TYPE
   DESCRIPTION
   LYON-STICKNEY-FIRE-SURVIVAL-STRATEGY
   NOBLE-AND-SLATYER-VITAL-ATTRIBUTES
     SPECIES-TYPE
     TIME-UNTIL-MATURITY
     TIME-UNTIL-SENESCENCE
     TIME-UNTIL-EXTINCTION
   ROWE-MODE-OF-PERSISTANCE
   REFERENCES
   ADAPTIVE-TRAITS-PARENT)
  (frame)
  :gettable-instance-variables
  :settable-instance-variables)

(defmethod
     (fire-adaptive-traits-and-survival-strategies
      :SLOT-LIST)
     ()
 '(FRAME-TYPE
   DESCRIPTION
   LYON-STICKNEY-FIRE-SURVIVAL-STRATEGY
   NOBLE-AND-SLATYER-VITAL-ATTRIBUTES
     SPECIES-TYPE
     TIME-UNTIL-MATURITY
     TIME-UNTIL-SENESCENCE
     TIME-UNTIL-EXTINCTION
   ROWE-MODE-OF-PERSISTANCE
   REFERENCES
   ADAPTIVE-TRAITS-PARENT))

(defmethod (fire-adaptive-traits-and-survival-strategies
             :QUERY-DISPLAY-SLOT-LIST)
            ()
 '(DESCRIPTION
   LYON-STICKNEY-FIRE-SURVIVAL-STRATEGY
   NOBLE-AND-SLATYER-VITAL-ATTRIBUTES
     SPECIES-TYPE
     TIME-UNTIL-MATURITY
     TIME-UNTIL-SENESCENCE
     TIME-UNTIL-EXTINCTION
   ROWE-MODE-OF-PERSISTANCE
   REFERENCES))
```

```
(defmethod (fire-adaptive-traits-and-survival-strategies
            :QUERY-DISPLAY)
           ()
  (send self
     :query-species-print-frame-header-utility
     "Fire Adaptive Traits and Survival Strategies
      Information")
  (let* ((slot-list (send self :QUERY-DISPLAY-SLOT-LIST))
      (display-list
       (do ((slot-list slot-list (cdr slot-list))
            (displayable-list
          nil
          (cond ((and (eq (car slot-list)
                           'NOBLE-AND-SLATYER-VITAL-ATTRIBUTES)
                       (or (not (eq (get-data-frame-slot
                                      'self
                                      'SPECIES-TYPE)
                                    'no-entry))
                           (not (eq (get-data-frame-slot
                                      'self
                                      'TIME-UNTIL-MATURITY)
                                    'no-entry))
                           (not (eq (get-data-frame-slot
                                      'self
                                      'TIME-UNTIL-SENESCENCE)
                                    'no-entry))
                           (not (eq (get-data-frame-slot
                                      'self
                                      'TIME-UNTIL-EXTINCTION)
                                    'no-entry)))))
                     (cons 'NOBLE-AND-SLATYER-VITAL-ATTRIBUTES
                        displayable-list))
                    ((eq (get-data-frame-slot
                           'self
                           (car slot-list))
                          'no-entry)
                        displayable-list)
                    (t (cons (car slot-list)
                                  displayable-list)))))
            ((null slot-list) (reverse displayable-list)))))
     (cond ((null display-list)
            (print-slot
          '(NL
            "Sorry, no information available on this
             subject!"
            NL)
          "text"))
          (t (do ((display-list display-list
                                (cdr display-list)))
                 ((null display-list) nil)
                 (send
                  (send self (find-symbol
                              (string (car display-list))
                              *keyword-package*))
                  :display)))))
  (readcontinue)
  t)

(defmethod (fire-adaptive-traits-and-survival-strategies
            :SPECIES)
           ()
  (send
   (eval (get-data-frame-slot 'self
                              'ADAPTIVE-TRAITS-PARENT))
   :SPECIES))


;************************************************************

;FIRE-EFFECTS Frame definitions

;************************************************************

(defflavor fire-effects
  (FRAME-TYPE
   FIRE-EFFECT-ON-PLANT
   DISCUSSION-AND-QUALIFICATION-OF-FIRE-EFFECT
   PLANT-RESPONSE-TO-FIRE
   DISCUSSION-AND-QUALIFICATION-OF-PLANT-RESPONSE
   SEVERITY-SEASON-SPECIFIC-FIRE-EFFECTS
   REFERENCES
   FIRE-EFFECTS-PARENT)
  (frame)
  :gettable-instance-variables
  :settable-instance-variables)

(defmethod (fire-effects :SLOT-LIST) ()
 '(FRAME-TYPE
   FIRE-EFFECT-ON-PLANT
   DISCUSSION-AND-QUALIFICATION-OF-FIRE-EFFECT
   PLANT-RESPONSE-TO-FIRE
   DISCUSSION-AND-QUALIFICATION-OF-PLANT-RESPONSE
   SEVERITY-SEASON-SPECIFIC-FIRE-EFFECTS
   REFERENCES
   FIRE-EFFECTS-PARENT))
```

```
(defmethod (fire-effects :QUERY-DISPLAY-SLOT-LIST) ()
 '(FIRE-EFFECT-ON-PLANT
   DISCUSSION-AND-QUALIFICATION-OF-FIRE-EFFECT
   PLANT-RESPONSE-TO-FIRE
   DISCUSSION-AND-QUALIFICATION-OF-PLANT-RESPONSE
   REFERENCES))

(defmethod (fire-effects :QUERY-DISPLAY) ()
  (send self :query-view-frame-utility
     'query-species-print-frame-header-utility
     "Fire Effects Information")
  (let ((sssfe-list (get-data-frame-slot
		       'self
		       'SEVERITY-SEASON-SPECIFIC-FIRE-EFFECTS)))
       (cond ((not (eq sssfe-list 'no-entry))
	   (detailed-fire-effects-menu sssfe-list)))))

(defmethod (fire-effects :SPECIES) ()
  (send
   (eval (get-data-frame-slot 'self 'FIRE-EFFECTS-PARENT))
   :SPECIES))

;************************************************************

;SEVERITY-SEASON-SPECIFIC-FIRE-EFFECTS Frame definitions

;************************************************************

(defflavor severity-season-specific-fire-effects
  (FRAME-TYPE
   SEVERITY
   SEASON
   EFFECT
   CERTAINTY-FACTOR
   DESCRIPTION
   QUALIFICATION
   REFERENCES
   FIRE-EFFECT-PARENT)
  (frame)
  :gettable-instance-variables
  :settable-instance-variables)
```

```
(defmethod (severity-season-specific-fire-effects
            :SLOT-LIST)
           ()
 '(FRAME-TYPE
   SEVERITY
   SEASON
   EFFECT
   CERTAINTY-FACTOR
   DESCRIPTION
   QUALIFICATION
   REFERENCES
   FIRE-EFFECT-PARENT))

(defmethod (severity-season-specific-fire-effects
            :QUERY-DISPLAY-SLOT-LIST)
           ()
 '(SEVERITY
   SEASON
   EFFECT
   CERTAINTY-FACTOR
   DESCRIPTION
   QUALIFICATION
   REFERENCES))

(defmethod (severity-season-specific-fire-effects
            :QUERY-DISPLAY)
           ()
  (send self :query-view-frame-utility
     'query-species-print-frame-header-utility
     "Severity-Season Fire Effects Information")
  t)

(defmethod (severity-season-specific-fire-effects :SPECIES)
           ()
  (send
   (eval (get-data-frame-slot 'self 'FIRE-EFFECT-PARENT))
   :SPECIES))
```

FLAVOR AND METHOD DEFINITIONS FOR THE CREATION OF SLOT HIERARCHY SYSTEM FRAMES

```
;************************************************************

;Atom FRAME -- atom class slot type definitions

;************************************************************

(defflavor atom () ())
```

```
(defmethod (atom :display) ()
  (let ((display-list (cons (send self :name)
                    (cons ": "
                       (cons (send self :value)
                          (list 'NL 'NL))))))
       (print-slot display-list 'atom)))

(defmethod (atom :display-atom-subslot) ()
  (let ((display-list
                    (cons (string-append "  "
                                         (send self :name))
                      (cons ": "
                            (cons (send self :value)
                                  (list 'NL 'NL))))))
       (print-slot display-list 'atom)))

;*************************************************************

;Header FRAME -- header class slot type definitions

;*************************************************************

(defflavor header () ())

(defmethod (header :display) ()
  (let ((display-list (cons (send self :name)
                    (cons ": " (list 'NL 'NL)))))
       (print-slot display-list 'header)))


;*************************************************************

;List FRAME -- list class slot type definitions

;*************************************************************

(defflavor list () ())

(defmethod (list :display) ()
  (let ((display-list (cons (send self :name)
                    (cons ": "
                       (append (send self :value)
                            (list 'NL 'NL))))))
       (print-slot display-list 'list)))
```

```
(defmethod (list :display-list-subslot) ()
  (let ((display-list
          (cons (string-append "   " (send self :name))
                (cons ": "
                      (append (send self :value)
                              (list 'NL 'NL))))))
       (print-slot display-list 'list)))


;***********************************************************

;Text FRAME -- text class slot type definitions

;***********************************************************

(defflavor text () ())

(defmethod (text :display) ()
  (let ((display-list (cons (send self :name)
                  (cons ": "
                     (append (send self :value)
                          (list 'NL 'NL))))))
       (print-slot display-list 'text)))

(defmethod (text :display-text-subslot) ()
  (let ((display-list
          (cons (string-append "   " (send self :name))
                (cons ": "
                      (append (send self :value)
                              (list 'NL 'NL))))))
       (print-slot display-list 'text)))


;***********************************************************

;Generated pointer FRAMES
;-- Generated pointer class slot type definitions

;***********************************************************

(defflavor generated-frame-pointer () ())

(defflavor generated-frame-pointer-list () ())
```

```
;*************************************************************

;FRAME-TYPE FRAME -- FRAME-TYPE slot type definitions

;*************************************************************

(defflavor FRAME-TYPE (value
                         (type 'atom)
                         (name "FRAME TYPE"))
                        (atom)
             :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)


;*************************************************************

;SPECIES FRAME -- SPECIES slot type definitions

;*************************************************************

(defflavor SPECIES (value (type 'atom)(name "SPECIES"))
                   (atom)
             :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)


;*************************************************************

;SCIENTIFIC-ALIAS FRAME -- SCIENTIFIC-ALIAS slot type
;                          definitions

;*************************************************************

(defflavor SCIENTIFIC-ALIAS (value
                               (type 'list)
                               (name "SCIENTIFIC ALIAS"))
                              (list)
             :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)
```

```
;*********************************************************

;ABBREVIATION FRAME -- ABBREVIATION slot type definitions

;*********************************************************

(defflavor ABBREVIATION (value
                          (type 'atom)
                          (name "ABBREVIATION"))
                         (atom)
            :settable-instance-variables
            :gettable-instance-variables
            :inittable-instance-variables)

;*********************************************************

;COMMON-NAMES FRAME -- COMMON-NAMES slot type definitions

;*********************************************************

(defflavor COMMON-NAMES (value
                          (type 'list)
                          (name "COMMON NAMES"))
                         (list)
            :settable-instance-variables
            :gettable-instance-variables
            :inittable-instance-variables)

;*********************************************************

;LIFE-FORM FRAME -- LIFE-FORM slot type definitions

;*********************************************************

(defflavor LIFE-FORM (value (type 'atom)(name "LIFE FORM"))
                      (atom)
            :settable-instance-variables
            :gettable-instance-variables
            :inittable-instance-variables)
```

```
;*******************************************************

;VARIETIES-AND-FORMS FRAME -- VARIETIES-AND-FORMS slot type
;                              definitions

;*******************************************************

(defflavor VARIETIES-AND-FORMS
  (value (type 'text)(name "VARIETIES AND FORMS")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;*******************************************************

;DISTRIBUTION-AND-OCCURRENCE FRAME
;-- DISTRIBUTION-AND-OCCURRENCE slot type definitions

;*******************************************************

(defflavor DISTRIBUTION-AND-OCCURRENCE
  (value
   (type 'generated-frame-pointer)
   (pointer-to 'distribution-and-occurrence)
   (name "DISTRIBUTION AND OCCURRENCE"))
  (generated-frame-pointer)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;*******************************************************

;VALUE-AND-USE FRAME -- VALUE-AND-USE slot type definitions

;*******************************************************

(defflavor VALUE-AND-USE
  (value
   (type 'generated-frame-pointer)
   (pointer-to 'value-and-use)
   (name "VALUE AND USE"))
  (generated-frame-pointer)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)
```

```
;*************************************************************

;BOTANICAL-AND-ECOLOGICAL-CHARACTERISTICS FRAME
;-- BOTANICAL-AND-ECOLOGICAL-CHARACTERISTICS slot type
;                                            definitions

;*************************************************************

(defflavor BOTANICAL-AND-ECOLOGICAL-CHARACTERISTICS
  (value
   (type 'generated-frame-pointer)
   (pointer-to 'botanical-and-ecological-characteristics)
   (name "BOTANICAL AND ECOLOGICAL CHARACTERISTICS"))
  (generated-frame-pointer)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;*************************************************************

;FIRE-ADAPTIVE-TRAITS-AND-SURVIVAL-STRATEGIES FRAME
;-- FIRE-ADAPTIVE-TRAITS-AND-SURVIVAL-STRATEGIES
;   slot type definitions

;*************************************************************

(defflavor FIRE-ADAPTIVE-TRAITS-AND-SURVIVAL-STRATEGIES
  (value
   (type 'generated-frame-pointer)
   (pointer-to
      'fire-adaptive-traits-and-survival-strategies)
   (name "FIRE ADAPTIVE TRAITS AND SURVIVAL STRATEGIES"))
  (generated-frame-pointer)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)
```

```
;************************************************************

;FIRE-EFFECTS FRAME -- FIRE-EFFECTS slot type definitions

;************************************************************

(defflavor FIRE-EFFECTS
  (value
   (type 'generated-frame-pointer)
   (pointer-to 'fire-effects)
   (name "FIRE EFFECTS"))
  (generated-frame-pointer)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;************************************************************

;SUPERIOR-PARENT FRAME
;-- SUPERIOR-PARENT slot type definitions

;************************************************************

(defflavor SUPERIOR-PARENT
  (value
   (type 'generated-frame-pointer)
   (pointer-to 'superior)
   (name "SUPERIOR PARENT"))
  (generated-frame-pointer)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;************************************************************

;GENERAL-DISTRIBUTION FRAME
;-- GENERAL-DISTRIBUTION slot type definitions

;************************************************************

(defflavor GENERAL-DISTRIBUTION
  (value (type 'text)(name "GENERAL DISTRIBUTION")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)
```

```
;***********************************************************

;BLM-PHYSIOGRAPHIC-REGIONS FRAME
;-- BLM-PHYSIOGRAPHIC-REGIONS slot type definitions

;***********************************************************

(defflavor BLM-PHYSIOGRAPHIC-REGIONS
  (value (type 'list)(name "BLM PHYSIOGRAPHIC REGIONS"))
  (list)
             :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)

;***********************************************************

;KUCHLER-PLANT-ASSOCIATIONS FRAME
;-- KUCHLER-PLANT-ASSOCIATIONS slot type definitions

;***********************************************************

(defflavor KUCHLER-PLANT-ASSOCIATIONS
  (value (type 'list)(name "KUCHLER PLANT ASSOCIATIONS"))
  (list)
             :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)

;***********************************************************

;SAF-COVER-TYPES FRAME
;-- SAF-COVER-TYPES slot type definitions

;***********************************************************

(defflavor SAF-COVER-TYPES
  (value (type 'list)(name "SAF COVER TYPES")) (list)
             :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)
```

```
;*********************************************************

;HABITAT-TYPE-INFORMATION FRAME
;-- HABITAT-TYPE-INFORMATION slot type definitions

;*********************************************************

(defflavor HABITAT-TYPE-INFORMATION
  (value (type 'text)(name "HABITAT TYPE INFORMATION"))
  (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;*********************************************************

;SPECIES-HABITAT-TYPES FRAME
;-- SPECIES-HABITAT-TYPES slot type definitions

;*********************************************************

(defflavor SPECIES-HABITAT-TYPES
  (value (type 'text)(name "SPECIES HABITAT TYPES")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;*********************************************************

;REFERENCES FRAME -- REFERENCES slot type definitions

;*********************************************************

(defflavor REFERENCES
  (value (type 'list)(name "REFERENCES")) (list)
            :settable-instance-variables
            :gettable-instance-variables
            :inittable-instance-variables)
```

```
;***********************************************************

;DISTRIBUTION-PARENT FRAME
;-- DISTRIBUTION-PARENT slot type definitions

;***********************************************************

(defflavor DISTRIBUTION-PARENT
  (value
   (type 'generated-frame-pointer)
   (pointer-to 'species)
   (name "DISTRIBUTION PARENT"))
  (generated-frame-pointer)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;***********************************************************

;DESCRIPTION FRAME -- DESCRIPTION slot type definitions

;***********************************************************

(defflavor DESCRIPTION
  (value (type 'text)(name "DESCRIPTION")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;***********************************************************

;PALATABILITY FRAME -- PALATABILITY slot type definitions

;***********************************************************

(defflavor PALATABILITY
  (value (type 'text)(name "PALATABILITY")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)
```

```
;***********************************************************

;FOOD-VALUE FRAME -- FOOD-VALUE slot type definitions

;***********************************************************

(defflavor FOOD-VALUE
  (value (type 'text)(name "FOOD VALUE")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;***********************************************************

;COVER-VALUE FRAME -- COVER-VALUE slot type definitions

;***********************************************************

(defflavor COVER-VALUE
  (value (type 'text)(name "COVER VALUE")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;***********************************************************

;IMPORTANCE-TO-LIVESTOCK-AND-WILDLIFE FRAME
;IMPORTANCE-TO-LIVESTOCK-AND-WILDLIFE --  slot type
;                                          definitions

;***********************************************************

(defflavor IMPORTANCE-TO-LIVESTOCK-AND-WILDLIFE
           (value
            (type 'text)
            (name "IMPORTANCE TO LIVESTOCK AND WILDLIFE"))
           (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;***********************************************************

;OTHER-USES-AND-VALUES FRAME
;-- OTHER-USES-AND-VALUES slot type definitions
```

```
;**********************************************************

(defflavor OTHER-USES-AND-VALUES FRAME
  (value (type 'text)(name "OTHER-USES-AND-VALUES")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;**********************************************************

;ENVIRONMENTAL-CONSIDERATIONS FRAME
;-- ENVIRONMENTAL-CONSIDERATIONS slot type definitions

;**********************************************************

(defflavor ENVIRONMENTAL-CONSIDERATIONS
  (value (type 'text)(name "OTHER-USES-AND-VALUES")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;**********************************************************

;VALUE-AND-USE-PARENT FRAME
;-- VALUE-AND-USE-PARENT slot type definitions

;**********************************************************

(defflavor VALUE-AND-USE-PARENT
  (value
   (type 'generated-frame-pointer)
   (pointer-to 'species)
   (name "VALUE AND USE PARENT"))
  (generated-frame-pointer)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;**********************************************************

;GENERAL-DESCRIPTION FRAME
;-- GENERAL-DESCRIPTION slot type definitions

;**********************************************************

(defflavor GENERAL-DESCRIPTION
  (value (type 'text)(name "GENERAL DESCRIPTION")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)
```

```
;***********************************************************

;GROWTH-FORM FRAME
;-- GROWTH-FORM slot type definitions

;***********************************************************

(defflavor GROWTH-FORM
  (value (type 'list)(name "GROWTH FORM")) (list)
             :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)

;***********************************************************

;RAUNKIAER-LIFE-FORM FRAME
;-- RAUNKIAER-LIFE-FORM slot type definitions

;***********************************************************

(defflavor RAUNKIAER-LIFE-FORM
  (value (type 'list)(name "RAUNKIAER LIFE FORM")) (list)
             :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)

;***********************************************************

;GRIME-PLANT-STRATEGY-CLASSIFICATION FRAME
;-- GRIME-PLANT-STRATEGY-CLASSIFICATION slot type
;                                       definitions

;***********************************************************

(defflavor GRIME-PLANT-STRATEGY-CLASSIFICATION
           (value
            (type 'list)
            (name "GRIME PLANT STRATEGY CLASSIFICATION"))
           (list)
             :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)
```

```
;***********************************************************

;GRIME-REGENERATIVE-STRATEGY-CLASSIFICATION FRAME
;-- GRIME-REGENERATIVE-STRATEGY-CLASSIFICATION slot type
;                                                definitions

;***********************************************************

(defflavor GRIME-REGENERATIVE-STRATEGY-CLASSIFICATION
           (value
            (type 'list)
            (name "GRIME REGENERATIVE STRATEGY
                   CLASSIFICATION"))
           (list)
             :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)

;***********************************************************

;REGENERATION-PROCESSES FRAME
;-- REGENERATION-PROCESSES slot type definitions

;***********************************************************

(defflavor REGENERATION-PROCESSES
  (value (type 'text)(name "REGENERATION PROCESSES")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;***********************************************************

;SITE-CHARACTERISTICS FRAME
;-- SITE-CHARACTERISTICS slot type definitions

;***********************************************************

(defflavor SITE-CHARACTERISTICS
  (value (type 'text)(name "SITE CHARACTERISTICS")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)
```

```
;*************************************************************

;SUCCESSIONAL-STATUS FRAME
;-- SUCCESSIONAL-STATUS slot type definitions

;*************************************************************

(defflavor SUCCESSIONAL-STATUS
  (value (type 'text)(name "SUCCESSIONAL STATUS")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;*************************************************************

;SEASONAL-DEVELOPMENT FRAME
;-- SEASONAL-DEVELOPMENT slot type definitions

;*************************************************************

(defflavor SEASONAL-DEVELOPMENT
  (value (type 'text)(name "SEASONAL-DEVELOPMENT")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;*************************************************************

;BOTANICAL-CHARACTERISTICS-PARENT FRAME
;-- BOTANICAL-CHARACTERISTICS-PARENT slot type definitions

;*************************************************************

(defflavor BOTANICAL-CHARACTERISTICS-PARENT
  (value
   (type 'generated-frame-pointer)
   (pointer-to 'species)
   (name "BOTANICAL CHARACTERISTICS PARENT"))
  (generated-frame-pointer)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)
```

```
;*************************************************************

;LYON-STICKNEY-FIRE-SURVIVAL-STRATEGY FRAME
;-- LYON-STICKNEY-FIRE-SURVIVAL-STRATEGY slot type
;                                         definitions

;*************************************************************

(defflavor LYON-STICKNEY-FIRE-SURVIVAL-STRATEGY
  (value (type 'list)
     (name "LYON STICKNEY FIRE SURVIVAL STRATEGY"))
  (list)
            :settable-instance-variables
            :gettable-instance-variables
            :inittable-instance-variables)

;*************************************************************

;NOBLE-AND-SLATYER-VITAL-ATTRIBUTES FRAME
;-- NOBLE-AND-SLATYER-VITAL-ATTRIBUTES slot type
;                                       definitions

;*************************************************************

(defflavor NOBLE-AND-SLATYER-VITAL-ATTRIBUTES
  (value
   (type 'header)
   (name "NOBLE AND SLATYER VITAL ATTRIBUTES")) (header)
            :settable-instance-variables
            :gettable-instance-variables
            :inittable-instance-variables)

;*************************************************************

;SPECIES-TYPE FRAME -- SPECIES-TYPE slot type definitions

;*************************************************************

(defflavor SPECIES-TYPE
  (value (type 'list) (name "SPECIES TYPE")) (list)
            :settable-instance-variables
            :gettable-instance-variables
            :inittable-instance-variables)

(defmethod (SPECIES-TYPE :display) ()
  (send self :display-list-subslot))
```

```
;***********************************************************

;TIME-UNTIL-MATURITY FRAME
;-- TIME-UNTIL-MATURITY slot type definitions

;***********************************************************

(defflavor TIME-UNTIL-MATURITY
  (value (type 'atom) (name "TIME UNTIL MATURITY")) (atom)
             :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)

(defmethod (TIME-UNTIL-MATURITY :display) ()
  (send self :display-atom-subslot))

;***********************************************************

;TIME-UNTIL-SENESCENCE FRAME
;-- TIME-UNTIL-SENESCENCE slot type definitions

;***********************************************************

(defflavor TIME-UNTIL-SENESCENCE
  (value (type 'atom) (name "TIME UNTIL SENESCENCE")) (atom)
             :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)

(defmethod (TIME-UNTIL-SENESCENCE :display) ()
  (send self :display-atom-subslot))

;***********************************************************

;TIME-UNTIL-EXTINCTION FRAME
;-- TIME-UNTIL-EXTINCTION slot type definitions

;***********************************************************

(defflavor TIME-UNTIL-EXTINCTION
  (value (type 'atom) (name "TIME UNTIL EXTINCTION")) (atom)
             :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)

(defmethod (TIME-UNTIL-EXTINCTION :display) ()
  (send self :display-atom-subslot))
```

```
;***********************************************************

;ROWE-MODE-OF-PERSISTANCE FRAME
;-- ROWE-MODE-OF-PERSISTANCE slot type definitions

;***********************************************************

(defflavor ROWE-MODE-OF-PERSISTANCE
           (value
            (type 'list)
            (name "ROWE-MODE-OF-PERSISTANCE"))
           (list)
                   :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)

;***********************************************************

;ADAPTIVE-TRAITS-PARENT FRAME
;-- ADAPTIVE-TRAITS-PARENT slot type definitions

;***********************************************************

(defflavor ADAPTIVE-TRAITS-PARENT
  (value
   (type 'generated-frame-pointer)
   (pointer-to 'species)
   (name "ADAPTIVE TRAITS PARENT"))
  (generated-frame-pointer)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;***********************************************************

;FIRE-EFFECT-ON-PLANT FRAME
;-- FIRE-EFFECT-ON-PLANT slot type definitions

;***********************************************************

(defflavor FIRE-EFFECT-ON-PLANT
  (value (type 'text)(name "FIRE EFFECT ON PLANT")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)
```

```
;*************************************************************

;DISCUSSION-AND-QUALIFICATION-OF-FIRE-EFFECT FRAME
;-- DISCUSSION-AND-QUALIFICATION-OF-FIRE-EFFECT
;    slot type definitions

;*************************************************************

(defflavor DISCUSSION-AND-QUALIFICATION-OF-FIRE-EFFECT
  (value
   (type 'text)
   (name "DISCUSSION AND QUALIFICATION OF FIRE EFFECT"))
  (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;*************************************************************

;PLANT-RESPONSE-TO-FIRE FRAME
;-- PLANT-RESPONSE-TO-FIRE slot type definitions

;*************************************************************

(defflavor PLANT-RESPONSE-TO-FIRE
  (value (type 'text)(name "PLANT RESPONSE TO FIRE")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;*************************************************************

;DISCUSSION-AND-QUALIFICATION-OF-PLANT-RESPONSE FRAME
;-- DISCUSSION-AND-QUALIFICATION-OF-PLANT-RESPONSE
;    slot type definitions

;*************************************************************

(defflavor DISCUSSION-AND-QUALIFICATION-OF-PLANT-RESPONSE
  (value
   (type 'text)
   (name "DISCUSSION AND QUALIFICATION OF PLANT RESPONSE"))
  (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)
```

```
;***********************************************************

;SEVERITY-SEASON-SPECIFIC-FIRE-EFFECTS FRAME
;-- SEVERITY-SEASON-SPECIFIC-FIRE-EFFECTS
;    slot type definitions

;***********************************************************

(defflavor SEVERITY-SEASON-SPECIFIC-FIRE-EFFECTS
  (value
   (type 'generated-frame-pointer-list)
   (pointer-to 'severity-season-specific-fIre-effects)
   (name "SEVERITY SEASON SPECIFIC FIRE EFFECTS"))
  (generated-frame-pointer-list)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;***********************************************************

;FIRE-EFFECTS-PARENT FRAME
;-- FIRE-EFFECTS-PARENT slot type definitions

;***********************************************************

(defflavor FIRE-EFFECTS-PARENT
  (value
   (type 'generated-frame-pointer)
   (pointer-to 'species)
   (name "FIRE EFFECTS PARENT"))
  (generated-frame-pointer)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;***********************************************************

;SEVERITY FRAME -- SEVERITY slot type definitions

;***********************************************************

(defflavor SEVERITY
  (value (type 'atom) (name "SEVERITY")) (atom)
             :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)
```

```
;**********************************************************

;SEASON FRAME -- SEASON slot type definitions

;**********************************************************

(defflavor SEASON
  (value (type 'atom) (name "SEASON")) (atom)
             :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)

;**********************************************************

;EFFECT FRAME -- EFFECT slot type definitions

;**********************************************************

(defflavor EFFECT
  (value (type 'text) (name "EFFECT")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;**********************************************************

;CERTAINTY-FACTOR FRAME
;-- CERTAINTY-FACTOR slot type definitions

;**********************************************************

(defflavor CERTAINTY-FACTOR
  (value (type 'atom) (name "CERTAINTY-FACTOR")) (atom)
             :settable-instance-variables
             :gettable-instance-variables
             :inittable-instance-variables)

;**********************************************************

;DESCRIPTION FRAME
;-- DESCRIPTION slot type definitions

;**********************************************************

(defflavor DESCRIPTION
  (value (type 'text) (name "DESCRIPTION")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)
```

```
;*************************************************************

;QUALIFICATION FRAME
;-- QUALIFICATION slot type definitions

;*************************************************************

(defflavor QUALIFICATION
  (value (type 'text) (name "QUALIFICATION")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;*************************************************************

;FIRE-EFFECT-PARENT FRAME
;-- FIRE-EFFECT-PARENT slot type definitions

;*************************************************************

(defflavor FIRE-EFFECT-PARENT
  (value
   (type 'generated-frame-pointer)
   (pointer-to 'fire-effects)
   (name "FIRE EFFECT PARENT"))
  (generated-frame-pointer)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;*************************************************************

;INTRODUCTION FRAME
;-- INTRODUCTION slot type definitions

;*************************************************************

(defflavor INTRODUCTION
  (value
   (type 'generated-frame-pointer)
   (pointer-to 'introduction)
   (name "INTRODUCTION"))
  (generated-frame-pointer)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)
```

```
;*************************************************************

;SAGEBRUSH FRAME
;-- SAGEBRUSH slot type definitions

;*************************************************************

(defflavor SAGEBRUSH
  (value
   (type 'generated-frame-pointer)
   (pointer-to 'sagebrush)
   (name "SAGEBRUSH"))
  (generated-frame-pointer)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;*************************************************************

;SPECIES-LIST FRAME
;-- SPECIES-LIST slot type definitions

;*************************************************************

(defflavor SPECIES-LIST
  (value
   (type 'generated-frame-pointer-list)
   (pointer-to 'species)
   (name "SPECIES-LIST"))
  (generated-frame-pointer-list)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;*************************************************************

;SUPERIOR-PARENT FRAME
;-- SUPERIOR-PARENT slot type definitions

;*************************************************************

(defflavor SUPERIOR-PARENT
  (value
   (type 'generated-frame-pointer)
   (pointer-to 'superior)
   (name "SUPERIOR PARENT"))
  (generated-frame-pointer)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)
```

```
;**********************************************************

;SPECIES-INTRODUCTION FRAME
;-- SPECIES-INTRODUCTION slot type definitions

;**********************************************************

(defflavor SPECIES-INTRODUCTION
  (value (type 'text) (name "SPECIES INTRODUCTION")) (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;**********************************************************

;SAGEBRUSH-INTRODUCTION FRAME
;-- SAGEBRUSH-INTRODUCTION slot type definitions

;**********************************************************

(defflavor SAGEBRUSH-INTRODUCTION
           (value
            (type 'text)
            (name "SAGEBRUSH INTRODUCTION"))
           (text)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)

;**********************************************************

;INTRODUCTION-PARENT FRAME
;-- INTRODUCTION-PARENT slot type definitions

;**********************************************************

(defflavor INTRODUCTION-PARENT
  (value
   (type 'generated-frame-pointer)
   (pointer-to 'introduction)
   (name "INTRODUCTION PARENT"))
  (generated-frame-pointer)
  :settable-instance-variables
  :gettable-instance-variables
  :inittable-instance-variables)
```

INTERFACE FUNCTION DEFINITION

```
*** The following Franz Lisp Function definition         ***
*** implements the former 'get-data-frame-slot' function ***
*** so that it utilizes the Flavors 'send' function and  ***
*** thereby provides message passage capability.         ***

(defun get-data-frame-slot (frame-pointer slot-name)
  (let* ((frame-slot-value
      (send (eval frame-pointer)
            (find-symbol (string slot-name)
                   *keyword-package*)))
      (slot-value (cond ((and (atom frame-slot-value)
                      (not (symbolp frame-slot-value)))
                   (send frame-slot-value :value))
                  (t frame-slot-value))))
     (cond ((null slot-value) 'no-entry)
           (t slot-value))))
```

KNOWLEDGE BASE CONVERSION UTILITY

```
*** The following Franz Lisp Function definition      ***
*** provides a utility for the conversion of original ***
*** knowledge base frame structures into flavors      ***
*** instances.                                        ***

(defun instantiate (list)
  (do ((list list (cdr list)))
      ((null list) t)
      (cond ((string= (subseq (string (car list)) 0 2) "sp")
          (set (car list) (make-instance 'species))
          (let ((frame-pointer (car list))
             (slot-list (get 'species/metaframe
                             'SLOT-LIST)))
            (do ((list slot-list (cdr list)))
                ((null list) t)
                (let ((slot-pointer (make-instance
                                      (car list))))
                  (send (eval frame-pointer)
                     (find-symbol
                      (string (concat "set-" (car list)))
                      *keyword-package*)
                     slot-pointer)
                (send slot-pointer
                   :set-value
                   (get frame-pointer (car list)))))))
```

```
((string= (subseq (string (car list)) 0 5) "distr")
 (set (car list)
      (make-instance 'distribution-and-occurrence))
 (let ((frame-pointer (car list))
    (slot-list
      (get 'distribution-and-occurrence/metaframe
           'SLOT-LIST)))
   (do ((list slot-list (cdr list)))
       ((null list) t)
       (let ((slot-pointer (make-instance
                                  (car list))))
         (send (eval frame-pointer)
            (find-symbol
             (string (concat "set-" (car list)))
             *keyword-package*)
            slot-pointer)
       (send slot-pointer
          :set-value
          (get frame-pointer (car list)))))))
((string= (subseq (string (car list)) 0 4) "mgmt")
 (set (car list) (make-instance 'value-and-use))
 (let ((frame-pointer (car list))
    (slot-list (get 'value-and-use/metaframe
             'SLOT-LIST)))
   (do ((list slot-list (cdr list)))
       ((null list) t)
       (let ((slot-pointer (make-instance
                                  (car list))))
         (send (eval frame-pointer)
            (find-symbol
             (string (concat "set-" (car list)))
             *keyword-package*)
            slot-pointer)
       (send slot-pointer
          :set-value
          (get frame-pointer (car list)))))))
((string= (subseq (string (car list)) 0 3) "bot")
 (set (car list)
    (make-instance
       'botanical-and-ecological-characteristics))
 (let ((frame-pointer (car list))
    (slot-list
     (get 'botanical-and-ecological-
           characteristics/metaframe
          'SLOT-LIST)))
      (do ((list slot-list (cdr list)))
          ((null list) t)
          (let ((slot-pointer
                   (make-instance (car list))))
            (send (eval frame-pointer)
               (find-symbol
                (string (concat "set-" (car list)))
                *keyword-package*)
               slot-pointer)
          (send slot-pointer
             :set-value
             (get frame-pointer (car list)))))))
  ((string= (subseq (string (car list)) 0 5) "adapt")
   (set (car list)
     (make-instance
      'fire-adaptive-traits-and-survival-strategies))
   (let ((frame-pointer (car list))
       (slot-list
        (get
         'fire-adaptive-traits-and-survival-
         strategies/metaframe
         'SLOT-LIST)))
      (do ((list slot-list (cdr list)))
          ((null list) t)
          (let ((slot-pointer
                   (make-instance (car list))))
            (send (eval frame-pointer)
               (find-symbol
                (string (concat "set-" (car list)))
                *keyword-package*)
               slot-pointer)
          (send slot-pointer
             :set-value
             (get frame-pointer (car list)))))))
  ((string= (subseq (string (car list)) 0 3) "gfe")
   (set (car list) (make-instance 'fire-effects))
   (let ((frame-pointer (car list))
       (slot-list
        (get 'fire-effects/metaframe 'SLOT-LIST)))
      (do ((list slot-list (cdr list)))
          ((null list) t)
          (let ((slot-pointer
                   (make-instance (car list))))
            (send (eval frame-pointer)
               (find-symbol
                (string (concat "set-" (car list)))
                *keyword-package*)
               slot-pointer)
          (send slot-pointer
             :set-value
             (get frame-pointer (car list)))))))
```

```
((string= (subseq (string (car list)) 0 5) "sssfe")
 (set (car list)
   (make-instance
      'severity-season-specific-fire-effects))
 (let ((frame-pointer (car list))
    (slot-list
     (get
        'severity-season-specific-fire-
         effects/metaframe
        'SLOT-LIST)))
   (do ((list slot-list (cdr list)))
       ((null list) t)
       (let ((slot-pointer
               (make-instance (car list))))
         (send (eval frame-pointer)
            (find-symbol
             (string (concat "set-" (car list)))
             *keyword-package*)
            slot-pointer)
       (send slot-pointer
          :set-value
          (get frame-pointer (car list)))))))
((eq (car list) 'superiorl)
 (set (car list)
   (make-instance 'superior))
 (let ((frame-pointer (car list))
    (slot-list
     (get 'superior/metaframe
       'SLOT-LIST)))
   (do ((list slot-list (cdr list)))
       ((null list) t)
       (let ((slot-pointer
               (make-instance (car list))))
         (send (eval frame-pointer)
            (find-symbol
             (string (concat "set-" (car list)))
             *keyword-package*)
            slot-pointer)
       (send slot-pointer
          :set-value
          (get frame-pointer (car list)))))))
((eq (car list) 'introl)
 (set (car list)
   (make-instance 'introduction))
   (let ((frame-pointer (car list))
      (slot-list
       (get 'introduction/metaframe
          'SLOT-LIST)))
     (do ((list slot-list (cdr list)))
         ((null list) t)
         (let ((slot-pointer
                  (make-instance (car list))))
           (send (eval frame-pointer)
              (find-symbol
               (string (concat "set-" (car list)))
               *keyword-package*)
              slot-pointer)
         (send slot-pointer
            :set-value
            (get frame-pointer (car list)))))))
)))
```

REFERENCES

Alexander, James H., "Smalltalk-80 aids troubleshooting system development", Systems & Software, Vol. 4, No. 4, p. 111-118, April 1985.

Alws, Karl-Heinz, Glasner-Schapeler, Ingrid, "EXPERIENCES WITH OBJECT ORIENTED PROGRAMMING", Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Vol. 2, p. 435-452, Springer-Verlag, Berlin, Germany, March, 1985.

Barbuceanu, Mihai, "Object-Centered representation and Reasoning: An Application to Computer-Aided Design", SIGART NEWSLETTER, No. 87, p. 33-39, January, 1984.

Baroody, A. James, DeWitt, David J., "An Object-Oriented Approach to Database System Implementation", ACM Transactions on Database Systems, Vol. 6, No. 4, p. 576-601, December, 1981.

Bobrow, Daniel G., Stefik, Mark J., "Perspectives on Artificial Intelligence Programming", SCIENCE, Vol. 231, p. 951-963, February 28, 1986.

Brown, Chris, "Programming language adds flexibility for artigicial intelligence", COMPUTER DESIGN, p. 28-30, June, 1983.

Buzzard, G. D., Mudge, T. N., "Object-Based Computing and the Ada Programming Language", IEEE Computer, Vol. 18, No. 3, p. 11-19, March, 1985.

Charniak,E.,McDermott, D., Introduction to Artificial Intelligence, Addison-Wesely, Reading, Massachusetts, 1985.

Cohen, A. Toni, "Data abstraction, data encapsulation and object-oriented programming", SIGPLAN Notices, Vol. 19, No. 1, p. 31-35, Janurary, 1984.

Cox, Brad J., "MESSAGE/OBJECT PROGRAMMING: AN EVOLUTIONARY CHANGE IN PROGRAMMING TECHNOLOGY", IEEE Software, Vol. 1, No.1, p. 50-61, January, 1984.

Cox, Brad, Hunt, Bill, "Objects, Icons, and Software-ICs", BYTE, Vol. 11 , No. 9, p. 161-176, August, 1986.

Fikes, Richard E., "A Knowledge-Based Assistant", Artificial Intelligence, Vol. 16, No. 3, p. 331-361, July, 1981.

Franz Inc., Franz Lisp Reference Manual, Franz Lisp Opus 42.16.3, Franz Inc., 1985.

Goldberg, Adele, Robson, David, SMALLTALK-80: The Language and its Implementation, Addison-Wesley, Reading, Massachusetts, 1983.

Greiner, Russell, "RLL-1: A Representational Language Language", Stanford Heuristic Programming Project, HPP-80-9 (Working Paper), Computer Science Department, Stanford University, Stanford CA, October 1980.

Hayes-Roth, F., Waternam, D. A., Lenat, D. B., (Ed's), Building Expert Systems, Addison-Wesely, Reading, Massachusetts, 1983.

Ingalls, Daniel H. H., "Design Principles Behind Smalltalk", BYTE, Vol. 6, No. 8, p. 286-298, August, 1981.

Leiberman, Heney, "Machine Tongues IX: Object-Oriented Programming", Computer Music Journal, Vol. 6, No. 3, p. 8-21, Fall 1982.

Lubinski, Thomas, Hutzel, Ingeborg, "An Object-Oriented Graphical Kernel System: The Birth of a Powerful Application Development Tool", Computer Graphics World, Vol. 17, No. 7, p. 69-74, July, 1984.

Martin, James, McClure, Carma, SOFTWARE MAINTENANCE: The Problem and its Solutions, PRENTICE-HALL,Inc., Englewood Cliffs, New Jersey, 1983

Pascoe, Geoffery, "Elements of Object-oriented Programming", BYTE, Vol. 11 , No. 9, p. 139-144, August, 1986.

Pratt, Terrence, PROGRAMMING LANGUAGES, PRENTICE-HALL, Inc., Englewood Cliffs, New Jersey, 1984.

Rentsch, Tim, "OBJECT ORIENTED PROGRAMMING", SIGPLAN Notices, Vol 17, No.9,p. 51-57, Sept. 1982.

Robson, David, "Object-Oriented Software Systems", BYTE, Vol. 6, No. 8, p. 74-86, August, 1981.

Schmucker, Kurt, J., "Object-oriented Languages for the Macintosh", BYTE, Vol. 11 , No. 9, p. 177-185, August, 1986.

Sheil, Beau, "Next-generation Software", IEEE Spectrum, Vol. 20, No. 11, p. 93, November, 1983.

Sprague, Richard, "Illuminating Objects", Macworld, p. 90-93, August, 1985.

Stefik, Mark, Bobrow, Daniel G., "Object-Oriented Programming: Themes and Variations", The AI MAGAZINE, Vol. 6, No. 4, p. 40-64, Winter, 1986.

Stoyan, Herbert, "What is an "Objekt-Oriented" Programming Language? Criteria for "object oriented" Programming Languages", Proceedings of the Seventeenth Annual Hawaii International Conference on System Sciences, Vol. 1, p. 152-62, 1984.

Tyugu, Enn H., "NUT - AN OBJECT ORIENTED LANGUAGE", Artificial Intelligence and Information-Control Systems of Robots, Elsevier Science Publishers B. V. (North-Holland), 1984.

White, Eva, Malloy, Rich, "Object-oriented Programming" BYTE, Vol. 11 , No. 9, p. 137, August, 1986.

Williams, Gregg, "Software Frameworks", BYTE, Vol. 9, No. 13, p. 124-127 & 394-410, December, 1984.

Winston, P. H., Horn, B. H., LISP, Addison-Wesley, Reading, Massachusetts, 1981.