THE REPRESENTATION OF PROGRAMS IN THE PROCEDURAL SEMANTIC NETWORK FORMALISM

by

Bryan M. Kramer

Technical Report CSRG-116

August, 1980
THE REPRESENTATION OF PROGRAMS IN THE PROCEDURAL SEMANTIC NETWORK FORMALISM

by

Bryan M. Kramer

Technical Report CSRG-116

August, 1980

The Computer Systems Research Group (CSRG) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is jointly administered by the Department of Electrical Engineering and the Department of Computer Science of the University of Toronto, and is supported in part by the National Research Council of Canada.
The Representation of Programs in the Procedural Semantic Network Formalism

by

Bryan M. Kramer

Department of Computer Science
University of Toronto

A thesis submitted in conformity with the requirements for the Degree of Master of Science at the University of Toronto

January 1990
Acknowledgements

I would like to thank Professor John Mylopoulos for his supervision of my work towards this thesis. I would also like to thank Professor Ray Perrault, my second reader, for his valuable comments on the nature of the manuscript. Finally, I would like to thank the members of the PSN group. The weekly meetings provided much needed criticism for new ideas.

Generous support was provided by the Ontario government, the Department of Computer Science, and the Natural Sciences and Engineering Research Council of Canada.
This thesis considers the interaction of programs with a semantic network formalism for organizing knowledge, and continues with a description of a computer implementation of the procedural semantic network formalism described by H. Levesque in "A Procedural Approach to Semantic Networks" (Technical Report No. 105, Department of Computer Science, University of Toronto). Programs in the formalism are objects examinable from the formalism, and are important because they are used to provide the semantics of classes. This thesis provides tools which further help the representation; in particular, it introduces a tool for organizing the structure of classes and uses this mechanism to organize the parts of programs. The implementation provides a translator which transforms an input language into objects within the network, and an interpreter which executes these objects as programs.
Contents

I Introduction
  1. Overview
  2. Background
  3. Outline

II Description of PSN
  1. Introduction
  2. INSTANCE-OF
  3. PART-OF
  4. IS-A
  5. Relations

III Metastructure and Programs
  1. Introduction
  2. Metastructure
  3. Programs
    3.1 Introduction
    3.2 The Structure of Programs
    3.3 Forms
    3.4 Sequential Program Constructs
  4. Exceptions

IV Implementation - Overview and Primitives
  1. Overview
  2. Storage Representation of PSN Objects
    2.1 Standard Objects
    2.2 Storing Classes
    2.3 Storing and Accessing Property Values
    2.4 Storing Relation Instances
  3. Creating objects
    3.1 Objects
    3.2 Adding Structure
# Contents

1. Introduction ... 116
2. Names and Constants ... 116
3. Expressions ... 117
4. Statements ... 119
5. Creation ... 121

Index ... 123
CHAPTER I

Introduction

The procedural semantic network formalism is a tool for representing and organizing the large amounts of knowledge often required by the solutions to problems in artificial intelligence. This thesis describes a computer implementation of the formalism and provides a new declarative description of programs within the model.

The philosophy of the procedural semantic network formalism (PSN) is to provide a small kernel of primitive functions and a set of declarative constructs all of which are expressible in terms of the kernel. It is natural therefore, to seek a representation of programs which uses the existing model. Although such representations have been proposed in older versions of PSN, there has been some room for improvement, especially in the representation of expressions (or statements) in a program.

The computer implementation of PSN incorporates this new model of programs in addition to the standard features of the formalism. Previous implementations have been less complete in that the procedures defining the semantics of PSN objects have been LISP code. Also, these systems supplied little more than the basic kernel of PSN. The new implementation provides a language which includes many of the declarative constructs, thus providing a more habitable environment for a user of the system.
I.1. Overview

The PSN formalism consists of semantic network primitives for representing knowledge, and a language in which changes in the knowledge represented can be expressed. The basic primitive is the object; the most primitive transformation of the knowledge base is the creation of a new object as in

"John := new object end".

The name "John" is an external name for the new object and can be used in the language or in text to refer to this object.

Objects can be related through the relation INSTANCE-OF as follows:

"make John instanceof PERSON".

The object referred to by "John" is now what is known as an instance of the class "PERSON". Classes play a very important role in the formalism; in addition to providing a means of separating objects in various categories, classes specify their behaviour through four attached programs. In the above example, the transformation which makes "John" an instance of "PERSON" is specified by the to-add program of the class. It is this procedural attachment which gives the formalism its name and makes programs an important area of study with respect to PSN.

Included in the definition of a class is a mechanism for defining structural properties which an instance of the class may have. The structural properties of an object are properties which the user of the formalism decides are essential to the description of the object. For example, each person could have as structural properties a sex and an eye colour. The definition of "PERSON" would then be

"PERSON := new CLASS
    structure
        sex := a SEX_VALUE end slot
        eye_colour := a COLOUR end slot
    end new"
and instances of this class could be assigned values for these properties as in

"John $. eye_colour <- blue".

For each individual property definition (for example sex) an object called a slot is created and associated with the class. This set of slots is known as the structure of the class. The class following the "a" in the slot definition is known as the type of the slot and is used to restrict the values which may be assigned for that property. If "blue" were not an instance of "COLOUR", the assignment illustrated above would fail. In general, a property value must be an instance of the class which is the type of the slot defining that property.

The first problem to which this thesis addresses itself is the organization of the slots defined in a class. In particular, one would like to group these slots (property definitions) into different categories. The organizing mechanism for structure is known as metastructure. This mechanism provides

1. the ability to distinguish categories of slots in the instance class; that is, where previously one could only ask for all of the slots of a class, one can now fetch all slots in the class which belong to a given category. This mechanism finds useful application in the representation of programs proposed in this thesis.

2. the ability to define properties of a slot. In particular, the type of a slot can be explained as a property of the slot.

3. the ability to specify a minimum and maximum number of instances of a given category which may appear in a class.

In effect, metastructure constrains the structure of classes while the structure of a class constrains the property values of instances of that class.
The motivation for the work on programs begins with a second important relation of PSN. The relation IS-A is a map between two classes, one of which becomes known as a subclass and the other as a superclass or IS-A parent. The relation is characterized as follows:

1. each instance of the subclass must be an instance of the superclass, and
2. all of the structure of the superclass is contained in the structure of the subclass.

The first property has important consequences for programs: as mentioned earlier, an object is made an instance of a class by the to-add program of the class. Now if, for example, "STUDENT" is a subclass of "PERSON", the add program of "STUDENT" must insure that an object added to the class is also made an instance of "PERSON". Thus, if

```
"STUDENT := new CLASS isa PERSON;
Mary := new STUDENT"
```

were executed, "Mary" should be an instance of both "PERSON" and "STUDENT".

The direction taken here is to allow IS-A to relate programs (by representing programs as classes) and insisting that the programs of a class be IS-A parents of the corresponding programs of a subclass of that class. The means of associating actions with a class in using it to represent a program must be such that IS-A related programs act so that the first condition for IS-A is satisfied.

The solution proposed in this thesis does not completely insure this requirement. What is provided, however, is a set of declarative rules which must be satisfied for one program to be an IS-A child of another, and an operational specification of how the actions of two such programs must be related. The latter is accompanied by guidelines which indicate when the constraints of IS-A would be violated. This is done in terms of inheritance: if one program IS-A another, it inherits the
actions of its parent. Thus if

"program1 := program ... end program;
program2 := program ... isa program1 end program"

(where the "...") indicates the actions of each program) were in effect, all of the actions of "program1" are inherited by "program2", and "program2" is legally an IS-A descendant of "program1" only if no action which is supplied in addition to those inherited undoes the effects of an inherited action.

The final problem for which this thesis proposes a solution is a computer implementation of the PSN formalism. This implementation will take as input commands in the language of which examples were shown above and perform the required changes in a knowledge base. The system must therefore provide a computer representation of the network structure. This includes a representation of objects, the IS-A and INSTANCE-OF relations, structure and metastructure, and procedural attachment. A translator is required to transform the input language into PSN objects which represent the commands which can then be interpreted by an interpreter of PSN programs.

The language implemented is based on that of the original description of PSN ([Levesque 1977]) with modifications for the new proposals of this thesis and to make the task of parsing simpler. The major PSN features of the language are constructs for creating objects and invoking the standard programs which are attached to classes. In addition it includes standard arithmetic and logical expressions and for, if, and begin control structures. The innovations involve the syntax for defining metastructures, for categorizing slots, and for specifying inheritance in programs.

In addition to providing a language in which transformations can be specified, an implementation must supply standard procedures which can be attached to the classes of a knowledge base. These will generally be used for what are known as stored or add-defined classes; such classes are
characterized by the fact that instances are identified by physically stored INSTANCE-OF links. The standard routines are primitive in that they are capable of creating and accessing such stored links. In contrast, test defined classes do not require the use of such routines. For example, one might define a class of blue eyed objects by associating with it a test routine which tests for an "eye_colour" equal to "blue". This test procedure would be written in the PSN language and does not involve the invocation of a standard routine.

Another aspect of the implementation is the provision of the inheritance mechanism. The major effort is in the inheritance of structure: this involves associating slots with classes, discovering which slots are to be inherited (insuring that no conflicting definitions are inherited), and adding the new properties to the new view of each slot. The inheritance of values (where a property value of a class is found by looking at its IS-A parents) involves finding all possible sources of inherited values and checking that a unique value may be determined.

The final aspect of the implementation is the interpreter. This program is used to interpret the PSN classes which are programs to produce the desired changes in the knowledge base. When a program is executed, an instance of the class which is the program is created. The concerns of the interpreter are the creation and termination of such instances, the association of the state of execution with these instances, and recording the relationships between such instances with the purpose of maintaining the dynamic order of execution.

An important concern in the design of the implementation is the final environment in which a user will find himself. This results in an interactive parser for the language which allows the user to correct errors as they are found through the use of special control tokens.
I.2 Background

The original description of PSN is found in [Levesque 1977]. This work also provides informally, the language which is a model for that used in the current implementation. [Levesque and Mylopoulos 1978] provides a good introduction to the formalism. Although it is a semantic network formalism, the traditional diagrams including nodes and arcs were not used in the original description of PSN. [Levesque and Mylopoulos 1978] introduces such a notation for the formalism.

An early use of PSN for the representation of a problem in artificial intelligence is described in [Kidd et al. 1977]. This project revealed several weaknesses in the formalism for which solutions were proposed in [Schneider 1978a]. These solutions are summarized in [Schneider 1978b]. This work improved somewhat the representation of programs, but subsequent evolution of the formalism has made the solution unusable.

An alternate stream of research originating from PSN has resulted in the language known as TAXIS ([Mylopoulos et al. 1978] and [Wong 1980]). This language is intended for the design of interactive information systems, taking from PSN primarily the organizational tools. These tools have been extended with much more concentration on the representation of programs than has been applied in PSN. The new representation of programs in PSN finds its roots in the programs of TAXIS. The fundamental differences between the version of PSN described in this thesis and TAXIS are:

1. PSN allows the dynamic creation and destruction of classes while TAXIS treats them as data types to be defined at compile time in the usual programming language sense.

2. PSN requires that expressions (that is, calls to programs as in \texttt{SQRT(5)}) be represented as objects within
the formalism. In T AXIS, the representation of an expression is an atomic (non-decomposable) unit on which no operation but evaluation is defined. One advantage of the PSN view is that the specialization of expressions is represented declaratively, while in T AXIS this can be expressed only in terms of a special predicate which returns true if one argument is a specialization of the second.

3. T AXIS uses what it calls property categories to organize the expressions of a program. These categories are not declaratively defined. PSN provides a general concept of metastructures which are definitions in a metaclass constraining the definitions of properties in classes.

4. T AXIS has a tool known as complex properties. Such properties find use in the exception handling mechanism. When the exception handling mechanism is adapted to PSN, complex properties are not available, and the representation becomes somewhat different from that in T AXIS.

5. In PSN execution of programs is accomplished by a program which interprets objects within the formalism. T AXIS executes compiled code.

The previous implementations of PSN are described in [Berlin 1978] and [Graham and Kramer 1979]. Experiments in the use of Berlin's system are described in [Lesperance 1979]. This work is a useful source of suggestions relevant to the design of an implementation originating from problems encountered in the use of such a system.
I.3. Outline

The current version of PSN is described in chapters two and three. Chapter two summarizes the features introduced in previous work and therefore serves as an introduction to the formalism. Chapter three introduces the new modifications to the model. This includes the introduction of metastructure and the new representation of programs.

The implementation of the model is discussed in chapters four and five. The standard procedures for storing, killing, testing, and fetching objects are described in chapter four. This corresponds roughly to what has been done in previous implementations. Chapter five continues with a description of the parser for the language and the interpreter of PSN programs.
II.1. Introduction

Although it is possible to work in PSN using only objects and programs operating on these objects, the formalism supplies several organizational tools designed to structure the task of representing knowledge. These declarative tools make explicit the relationships between various objects and kinds of knowledge in the knowledge base. This chapter summarizes the fundamental constructs of PSN: a way of categorizing objects by means of types, a way of associating properties with objects, and a means for specialization of types.

A PSN knowledge base in its basic form consists solely of objects some of which are procedures which provide the semantics for the objects to which they are attached. It is however convenient to consider the knowledge base from a higher declarative viewpoint. From this view the knowledge base shares features with other semantic network formalisms. In particular, the knowledge base consists of objects and relationships between these objects. When considered in this fashion it is natural to use a diagrammatic notation and vocabulary to describe features of the representation, thus the word node may often be used for object, and terms such as link or arc may be used to express the fact that a relationship exists between two objects.

There are three basic declarative relationships which characterize the PSN formalism. These are INSTANCE-OF which provides a means of giving objects types, PART-OF which gives objects structure and in particular explains procedural
II. 1 Introduction

attachment, and IS-A which provides a means of relating types. These tools interact through the inheritance mechanisms and thus provide a powerful means of describing knowledge.

The diagrammatic notation will be relied on heavily to provide illustrations of the declarative structures of the formalism. It will become necessary to refer to the objects and relationships expressed by the figures. Objects are represented as points or boxes usually accompanied by a label. This label is an example of an external name. In general, an external name is simply a means of referring to objects of some PSN knowledge base from the text; it is not a part of the formalism. Such references to objects will be enclosed in quotation marks ("""). For example, figure 2-1 illustrates objects referred to by "John" and "PERSON".

Diagrams will represent relationships between objects by arcs. In many cases the arcs will be labelled; however there are two types of arc which occur commonly and are not labelled. An unlabelled arc with a single shaft represents the relationship INSTANCE-OF described in section 2.2, and the unlabelled arc with a double shaft represents the IS-A relation described in section 2.4. Figure 2-1 includes an example of each of these arcs.

II. 2. INSTANCE-OF

Every object is related to at least one other object through the INSTANCE-OF relation. The set of objects associated to an object through this relation are known as the types of the object. Only objects whose types include the special object "CLASS" may be types of other objects. Such objects are known as classes; objects whose types include a given class are known as instances of that class. For example, the class "PERSON" would have as instances all objects in the knowledge base which are intended to represent people.
II.2 INSTANCE-OF

figure 2-1

figure 2-2
Classes play a very significant role in a PSN knowledge base. Through four programs which are attached to each class, the class determines the behaviour of its instances. In addition, as will be discussed in the next section, classes define the properties that an object may have. It is the attachment of the programs that gives the formalism its procedural aspect.

The action of asserting that a class is to become a type of an object is known as making the object an instance of the class. This operation is controlled by what is known as the to-add program of the class. This program will usually cause some kind of storage operation which will indicate that the object is an instance of the class. In addition, it can check that the object satisfies certain conditions for becoming a member of that class and can perform actions with side effects which are characteristic of instances of the class. For example, to implement two classes whose sets of instances are to be disjoint, the to-add programs of each class would check that new instances are not members of the other class.

The to-kill program of a class undoes the effects of the to-add program, that is, it modifies the knowledge base so that an object is no longer an instance of the class. Again, in addition to removing the link between object and class, this program may perform additional actions necessary for complete removal. The to-kill program, too, may check prerequisite conditions for removal and fail if they are not satisfied.

A third action associated with classes is testing for membership. This is the purpose of the to-test program. In its simplest form a to-test program will simply check for the existence of a link between an object and the class. In other cases, one might have classes whose instances are not explicitly linked to the class, and the to-test program would check other conditions for membership. For example, the program for a class "UNION_A_B" might return true if an
instance is a member of either class "A" or class "B".

The final program attached to classes is the to-fetch program whose responsibility is to make available all instances of a class. This program would be invoked in code such as

"for X in CLASS_A do <statements> end"

where as the iteration progresses, the variable "X" would take on the value of a different instance of "CLASS_A". In the example of "UNION_A_B" the to-fetch program could invoke the to-fetch programs of each of the classes "A" and "B" and return the set union of the results. Alternatively, a better algorithm is to fetch all the instances of one class and return those for which the test program of the other class returns true.

As indicated in the introduction, the INSTANCE-OF link is represented in diagrams as an unlabelled arrow with a single shaft. A class is often represented by a box to distinguish it from objects which are not classes. Procedural attachment to a class is indicated by the arcs labelled with the names of the respective programs as illustrated in figure 2-2. Since programs are part of the formalism, they are objects having types; in particular programs are objects which are instances of the class "PROGRAM". The details of the representation of programs will be discussed in chapter three. At this point it suffices to note that "PROGRAM", being a class, will have attached to it programs for the four operations which create, destroy, test for, and fetch programs.

Figure 2-2 also illustrates what is known as the INSTANCE-OF hierarchy. "John", "PERSON", and "CLASS" are objects at distinct levels in the representation known as the object, class, and meta levels respectively. A metaclass is a class whose instances are themselves classes. The need for an infinite number of levels where an object on one level is an instance of some object of the next higher level is obviated by making "CLASS" an instance of itself. This property of "CLASS"
is also made necessary by the rule which requires that types be instances of this object.

Another interesting object which must exist in a PSN knowledge base is the class "OBJECT" which has as instances any existing object. This includes objects from any level of the INSTANCE-OF hierarchy, and necessarily includes "OBJECT" itself.

II.3. PART-OF

Objects of the same type generally are grouped together because they share certain properties which characterize the class. For example, objects which represent people will certainly have properties describing their sex, eye colour, race, etc. PSN provides a mechanism for describing such properties of instances in the definition of the class. This description is known as the structure of the class. This structure is composed of a set of structural property definitions, commonly referred to as slots.

Corresponding to each slot there must be a structural property value (or slot value) for each instance of the class. Once assigned, a slot value may not be changed during the life of an object. In general, an object may be an instance of more than one class and therefore has slot values corresponding to the slots defined in each of its types.

A structural property definition serves three purposes. First, it specifies that there must be a corresponding relationship between an instance of the class and some other object. For example, a slot "S_I_N" in the class "PERSON" would indicate that each person has a social insurance number. Secondly, it specifies the types of which the slot value must be an instance. This set of classes is known as the type of the slot. The type of the slot "S_I_N" would include the class "NUMBER". The third aspect of a slot is the restriction which
is a predicate which further constrains the choice of a corresponding slot value. A restriction on the slot "$S_N$" might be that the number must have nine digits.

Another property which is associated with a slot is the default value. This value will be the slot value of an instance if, at the point at which the property is queried, no previous value has been assigned and no inherited (section 2.4 discusses inheritance) value was available.

The slots which make up the structure of a class are represented by PSN objects, and therefore may have types and property values. The class of which slots are instances is known as "slot". The objects which form the structure of a class are said to exist within that class. An object which exists inside a class is an object which has a unique association with the class; the relationship between such an object and its containing class is known as PART-OF. Objects which exist within a class must have among their types at least one object which is PART-OF a parent class of the class. There is here a potential ambiguity in the use of the word "type": when discussing objects it means the set of classes of which an object is an instance; when discussing slots it is a set constraining the corresponding slot values. Slots are however objects. In general it will be obvious which sense is intended.

In diagrams the structure of a class is represented by objects contained within the box representing the class. Objects which are instances of the class "slot" will generally be labelled; these names are used to mark the arcs joining instances of the class to values for the corresponding properties. This labelling is necessary only as a notational tool and says nothing of how the association is represented in a potential implementation.

The features type, restriction, and default which are
associated with a slot are simply property values which result from the structure defined for the class "slot" which exists in "CLASS". Thus these values are also linked to the slot by labelled arcs. However, as a notational convenience, types are represented by a set of arcs labelled "type" rather than the more consistent single arc joined to some representation of a set of classes.

As an illustration of these conventions, consider figure 2-3. This shows a class called "STUDENT" containing a single slot which represents a student number. "John" is an instance of student with number "741353839". The slot "number" has a single type; this is generally the case. In addition, "number" is an instance of "slot" which is contained in "CLASS". "slot" is not itself a slot, but it is an instance of "CLASS". As a class it may have instances (for example "number") and structure (the slots "type", "restriction", and "default"). The diagram shows the slots of this class as objects within the box representing "slot"; these slots are parts of "slot", not of "class". Since they are slots, these objects are instances of the class "slot" which is at the same time the class which contains them.

At this point in the discussion, no object has been introduced which has "slot" as an instance and is part of a type of "class". It will therefore become necessary to introduce a new class which will contain such an object. This will however be left until the discussion of metastructure in chapter three.

It is now possible to complete the explanation of procedural attachment to classes. The metaclass "CLASS" has as slots in its structure places for the four programs described earlier. An individual class may then have values for these slots. Should no value be provided, the default value may generally be used. Figure 2-4 illustrates the complete definition of "CLASS" and an example of a class. In this and
figure 2-3

figure 2-4
II.3 PART-OF

future diagrams the INSTANCE-OF link from a slot to "slot" is omitted unless it is necessary for clarity.

II.4. IS-A

Often in the construction of a knowledge base, one will find that it is necessary to define classes where the properties of one class are similar to those in another class. Differences might be on the restrictions on some slots: for example a "MAN" is a "PERSON" whose sex is restricted to "MALE". In other cases slots may be added: the class "STUDENT" may be like "PERSON" with the addition of a slot for a student number. In both these cases one speaks of one class being a specialization of the other.

In both the examples above, it is also true that an instance of the more specialized class is also an instance of the other class; a man is generally a person. These two properties characterize the PSN relation IS-A between two classes. If "A" IS-A "B" then "A" is a subclass of "B" and "B" is a superclass of "A". In the diagrams, IS-A is represented by an arrow with two shafts.

That an instance of a subclass must be an instance of the corresponding superclass must be reflected by the test and fetch programs of the classes. That requires, for example, that for every object for which the test program of "STUDENT" returns true, the test program of "PERSON" must also return true. The implication does not hold in the other direction. The fetch programs must be related so that the set of objects returned by the fetch program of the subclass must be contained in the set of objects returned by the fetch program of the superclass.

The passing of structure from a class to a subclass is an aspect of the process known as inheritance. A class "B" inherits from a superclass "A" any object which is defined in
"A" and is an instance of some object defined in some type of "B". In particular, if "B" IS-A "A", all the slots of "A" will be slots of "B". Figure 2-5 illustrates what may happen in inheritance of structure along IS-A. The slot "number" is shown only in "GRAD_STUDENT" because it is inherited without modification, but inheritance implies that it is a part of "PHD_STUDENT". The slot "supervisor" however undergoes a modification in inheritance. This does not imply the creation of a new object --- it is the same object seen in a different way. In modifying the properties of a slot one must obey what is known as the IS-A constraint. For "type" this constraint requires that any object which satisfies the types of the modified slot must satisfy the types of the original slot. This allows both the addition of new types and the specialization of existing types as "PROFESSOR" is a specialization of "PERSON" in the example. The slot "office" illustrates that new properties may be defined in the specialized class.

The IS-A constraint manifests itself differently for the various properties of slots. Since defaults will often not be classes, no constraint can be applied in inheritance because IS-A is not defined between arbitrary objects. The relationship between restrictions should be logical implication: if an object satisfies the restriction of the modified slot it should satisfy the restriction on the original slot. For other properties on slots, the general rule is that the value for the modified slot must be IS-A that of the original slot.

In addition to inheriting structure, a subclass may inherit property values from a superclass in the process of value inheritance. This is especially useful in cases where the subclass has some of its programs in common with its superclass. For example in figure 2-6 the class "STUDENT" differs from "PERSON" only through a different test program. Should one of the other programs be required in an operation on
figure 2-5

figure 2-6
students, the inheritance mechanism will find the appropriate program at the level of the class "PERSON" or higher if necessary. Inherited values are always preferred over the default value of a slot.

Value inheritance is pre-emptive; that is, if a value is explicitly specified it is preferred over any inherited value. Therefore, the consistency requirement on the four programs of a class is not constrained explicitly through the inheritance mechanism. One is tempted to create an IS-A constraint on the values of properties, so that, for instance, the test program of a class must be a specialization of the program of a superclass (it must be pointed out that the PSN definition of programs makes programs classes and hence, the IS-A relation holds between them). However, it is again possible to define properties of classes which will take on values which are not classes and for which specialization is not defined. Thus at present, value inheritance in PSN remains pre-emptive.

Consider the case where "A" IS-A "B" and "B" IS-A "C". Then an instance of "A" is an instance of "B" and therefore an instance of "C". The structure of "A" will contain slots inherited from "C" via "B" and any modifications of inherited slots will satisfy the IS-A constraint going from "C" to "A". Finally, any values which were not pre-empted in the definitions of "A" and "B" will be inherited from "C" to "A". Thus it is apparent that IS-A is transitive and "A" is an IS-A descendant of "C". The notion of transitivity also satisfies the English sense of the word "subclass": a subclass of a subclass of a class is obviously a subclass of the class.

Because IS-A is transitive, the resulting structure can legitimately be called a hierarchy. When the IS-A hierarchy presented so far is presented as a diagram, the result is a tree like structure. PSN, however does not constrain the hierarchy to be a tree. Thus a class may have IS-A parents which do not contain a common specialization among themselves.
This allows the possibility of conflicting inheritance of values and structure.

In the case of value inheritance there may be a set of values derived from immediate ancestors in the IS-A hierarchy, all of which are eligible. If these values are classes the problem is avoided if a member of the set is a specialization of all of the other classes. However, if the values are not classes or if there is no common specialization of a set of classes, no value can be inherited. In such cases of conflict, the solution presently taken by PSN, is to not inherit any value. An alternative solution is available when the IS-A hierarchy is extended to form a lattice as described in [Schneider 1978a]. In the case where a number of classes are inherited, the final value for the slot can be made the unique meet of these classes. If, however, not all of the inheritable values participate in the IS-A hierarchy, no value is inherited.

When structure is inherited from a number of superclasses, it is possible that two or more versions of a slot will be available. Figure 2-7 illustrates how conflicting versions of a slot may be inherited. The slot "major_place_of_work" is the same slot in each place it occurs although the properties of each version are slightly different. On the other hand, the slot "number" in "EMPLOYEE" is an object which is entirely unrelated to that in "STUDENT". In the latter case, there is no problem with inheritance within the formalism; the problem arises when one attempts to reference one of the inherited slots from the external language, and how it is handled lies in the domain of such a language, not within PSN. In English for example, one could distinguish the slots by qualifying the slot name with the class from which it was inherited. Thus, one might say "'number' as inherited from "EMPLOYEE"' when that is the slot one is referencing.

On the other hand, PSN must handle the inheritance of
figure 2-7

figure 2-8
conflicting versions of the same slot. A version of a slot is known as a redefinition of another version of the same slot if the latter version is part of the structure of some IS-A ancestor of the class whose structure contains the first version and some modification has occurred somewhere between the two slots. Thus in figure 2-8 "a" in "A1" is a redefinition of "a" in "A" whereas the inherited "a" in "A2" is not. Also, since the version of "a" in "A2" is the same as that in "A", "a" in "A1" is a redefinition of "a" in "A2". Now if the set of versions of a slot to be inherited contains a version which is a redefinition of all the other versions, that is, there is a unique redefinition, this version can be inherited as is without violation of the IS-A constraint.

In the case where no unique redefinition is contained in the inherited set, a new version of the slot must be created. This situation is very similar to that of value inheritance in that there is a set of values to be inherited by a version of a slot for each of the properties of a slot (for example type etc.). The solution is also the same: when all the values participate in the IS-A hierarchy, a meet class may be created; otherwise a value of unknown should be inherited.

II.5. Relations

The slot values of an object express relationships which are considered defining properties of that object. Objects may also participate in relationships that are more incidental: a man may have a wife, but being an instance of the class "Men" does not require that an object has a wife. Such properties of objects are known as assertional properties.

In PSN a class of objects known as relations are used to define relationships between objects. For example, one might define a relation called "Husband-of" which can be used to relate instances of "Men" to instances of "Women". A specific pair of objects are related through an assertion which is an
instance of the relation object describing the relationship. Thus "John" is related to "Mary" by an instance of "Husband-of". In this case "John" is the source of the assertion and "Mary" participates as the target. In diagrams, the connection between related objects is an arc; this arc is associated to the relation either by passing through a small circle connected to the relation by an INSTANCE-OF link, or by being labelled with the name of the relation. While the former expresses more clearly the flavour of the formalism, the latter is more useful because it avoids adding a proliferation of arcs to diagrams which are already very complicated. Figure 2-9 illustrates the relation "Husband-of" and the assertion between "John" and "Mary" using the more formal notation.

As with classes, the exact meaning of a relation is provided by four programs. Their functions correspond directly to those of classes: a program known as to-assert makes the knowledge that two objects are related by the relation part of the knowledge base; a testing program decides whether the relation holds between two objects; a fetching program returns all range instances of the relation corresponding to some object in the domain; and a removing program is used to destroy the relation instance.

The properties by which programs are attached to relations are defined as slots in the class "RELATION" as shown in figure 2-10. There are four additional property values attached to relations which the formalism uses to provide basic constraints on assertions. The "domain" property specifies a class of which any object participating in an assertion as the source must be a instance. The "range" property specifies a class which similarly constrains the target of an assertion. A relation is then a map from the "domain" class to the "range" class where the test program decides if a given pair of objects is a member of the relation.

A third additional property is the "domain_interval" which
II.5 Relations

figure 2-9

figure 2-10
is an interval constraining the number of assertions in which any object in the domain class may participate as the source. The interval \( <i,j> \) as a "domain_interval" indicates that each instance of the domain class is related to at least \( i \) and at most \( j \) objects in the range class. For example, the relation "Age" might have a "domain_interval" of \( <1,1> \) indicating that every object in its domain must have a unique age (its domain might be "REAL_OBJECTS"). This property differs from a slot "age" in the domain class in that it may be changed at any time.

Corresponding to the "domain_interval" (usually called "d_int") is the property "range_interval" ("r_int") which provides similar constraints on the number of assertions in which an instance of the range class may participate as the target.

It is natural to allow a specialization relation between relations, and PSN allows IS-A to relate two such objects. The meaning of IS-A can be adopted without change: an assertion is an instance of all superclasses of any relation of which it is an instance, and relations inherit values from their superrelations. (Relations will in general not have internal structure). In most cases, specialization of a relation requires the specialization of one or more of the eight property values. A simple example is restriction of a relation, where a subrelation in which the domain is a subclass of the inherited domain would be the restriction of the original relation to the new domain.

In PSN intervals are not represented as classes. Therefore, one can again not insist that the property values of relation be IS-A descendants of the inherited values. However, it is natural to consider a more restricted interval a specialization of a more general interval, that is, the lower end of the specialized interval is greater than that of its parent, and the higher end is less than that of the parent.
II.5 Relations

Thus $<3,5>$ is a specialization of $<1,8>$. It is necessary for the maintenance of consistency in the knowledge base that the intervals of a subrelation be specializations of the inherited intervals in this way.
CHAPTER III
Metastructure and Programs

III.1. Introduction

Since programs play a large role in PSN, it is useful to provide structuring tools which may help in the understanding of these objects. In particular one would like a declarative representation of programs which interacts with IS-A to provide a means of specialization. The approach taken is that of TAXIS [Wong 1980] in which the statements of a program are represented as properties of slots in a class. This approach requires that slots be differentiated, hence the concept of metastructure is introduced in section two.

Section three will discuss the structure of programs and the following sections will discuss the dynamic environment and the handling of exceptions.

III.2. Metastructure

In chapter two, the structure of a class was defined to be a set of objects which define the properties which an instance of the class may have. This section proposes a mechanism by which these slots may be organized. What results are objects which provide categories of slots just as classes provide categories of objects. In addition the mechanism will allow the definition of new categories of slots which have properties in addition to the standard "type", "default", and "restriction". A use for such additional properties is found in the representation of programs. Finally, the new mechanism provides a means for restricting the number of slots of a given category which may appear in a class.
This mechanism might be applied, for example, in the definition of a metaclass known as "PHYLUM" as part of a taxonomy of animals. A category of slots which must appear in each instance of this metaclass is a body part; for example the class "ARTHROPODA" would define the property "thorax" because each instance of this class would have a thorax. Restrictions on the set of body part slots might be that each instance of "PHYLUM" must define at least one body part, and that the type property of a body part slot have as value a class of body parts. Thus the type of the slot "thorax" would be the class "THORAX" which has as instances all individual thoraxes.

The solution begins by introducing the solution to a loose end left in chapter two. There it was pointed out that no class had been introduced which provided a part of which "slot" might be an instance. If it is made an instance of itself as is "CLASS", it would require for itself values for the properties "type", "restriction", and "default". The meaning of such values is not clear. In addition, "slot" would be treated differently from its other instances, for the other instances of "slot" are slots whereas "slot" cannot be considered a slot.

The alternative is to introduce a new class of which "CLASS" is an instance as shown in figure 3-1. This is the new class called "METACLASS" whose instances are all metaclasses and include, in particular, the metaclasses "CLASS" and "METACLASS". These two classes are instances of "CLASS" (as required) because they are instances of a subclass of "CLASS", namely "METACLASS". Since a metaclass is a class whose instances are classes, all metaclasses must ensure that their instances are in fact classes. The simplest way to enforce this is by requiring that all instances of "METACLASS" be subclasses of "CLASS". This is necessary also because instances of metaclasses may contain slot definitions only if the metaclass contains "slot" which it may acquire only through inheritance along IS-A. In the other direction, any subclass
III.2 Metastructure

**Figure 3-1**

**Figure 3-2**
of "CLASS" whose instances are to have structure, must be made an instance of "METACLASS" so that "slot" may be validly inherited.

"slot" is now an instance of "metaslot" which is an instance of "METACLASS" and therefore a subclass of "CLASS" and hence, since it is an instance of itself, an instance of "CLASS". Instances of "metaslot", or _metaslots_, are classes whose instances are slots and therefore impose constraints on and supply structure to these slots. Figure 3-2 illustrates an example of such a metaslot. Here it has been decided that instances of the metaclass "PHYLUM" should have slots which describe various body parts and that the types of these slots be limited to instances of the metaclass "BODY-PART-CLASS". This is done through the definition of the metaslot "body-part", which because it is a subclass of "slot" will inherit the properties of "slot". In particular, its instances will be slots, and its structure will contain the slot "type" for which the type is modified. Now slots in any instance of "PHYLUM" (for example, "ARTHROPODA") may be made instances of the new metaslot, suffering however the additional restriction on their type links.

It must be emphasized that the slot "type" illustrated in figure 3-2 is a slot of the class "body-part", not of the class "PHYLUM". Hence, this slot constrains the types of its instances; for example, the type of the slot "thorax" must be an instance of "BODY-PART-CLASS". The "interval" property value of "body-part" is <1,*>, thus requiring that any instance of "PHYLUM" have at least one instance of this class (the "*" is used to represent an infinite upper bound), as indeed, is the case in the class "ARTHROPODA".

"metaslot" contains within itself the slot "interval". The value of "interval" for a metaslot constrains the number of instances of that metaslot which may be PART-OF any individual instance of the defining metaclass. In figure 3-2, the
interval \(<1,\ast>\) for "body-part" means that each instance of "PHYLUM" must have at least one body part. The interval on "metaslot" is of course \(<0,\ast>\) allowing any number of metaslots in any metaclass.

Only if a metaslot is a subclass of "slot" will the instances of the metaslot be slots. Since PSN has no use (at present), for objects which are part of classes which are not either slots or metaslots, the add program of "metaslot" will force its instances to become subclasses of "slot" or of "metaslot". These subclasses of "slot" form a subset of the structure of a class which is known as the metastructure.

"METACLASS" is a metametaclass because its instances are metaclasses and "metaslot" is similarly a metametaslot. This has added explicitly this additional layer of the INSTANCE-OF levels which was previously only implicit. Higher levels are still possible because "METACLASS" is an instance of itself --- other metametaclasses can be defined as IS-A descendants of this class, and if instances of a subclass of "METACLASS" are also metametaclasses one has a class on a higher level. This can of course be continued indefinitely. The same thing is possible with "metaslot".

The notation thus far given for the definition of metaslots and the association of slots with their parent instances is far too cumbersome to use in most diagrams. Figure 3-3 illustrates the shorthands which are used for these two cases: any object drawn as a box within a box will represent a metaslot, thus be assumed an instance of "metaslot". In instances of a metaclass, the slots will be grouped in a list under the metaslot names. If no metaslot names are indicated, it is to be assumed that the slot is simply an instance of slot. Thus "MC1" is a metaclass in which the metaslots "ms1" and "ms2" are defined, and in the class "C1", "e" is an instance of "slot" only while "a", "b", and "c" are instances of "ms1" and "d" is an instance of "ms2".
figure 3-3
III.3 Programs

III.3.1 Introduction

The association of programs with classes is a major defining characteristic of PSN. This association is accomplished through the mechanism for associating property values with objects and assumes that programs are objects existing in the knowledge base. The next step is to involve these objects in the network in a way which allows the structure of programs to provide an indication of their functions and allows a declarative means of relating them.

The need for relating programs results from the IS-A hierarchy of classes: the relations between programs in the network will constrain what programs may be associated with IS-A related classes. Thus if "STUDENT" IS-A "PERSON", the programs for "PERSON" must be IS-A parents of the programs for "STUDENT". The following paragraphs will consider the relationships that IS-A between classes imposes on the four programs of each class.

When the add program for a class is executed, some change in the knowledge base will occur which indicates that an object is now an instance of the class. For example, an assertion of some relation may be created. A change in the knowledge base caused by the execution of a program is called a side effect. However, only the lasting side effects resulting from the execution of a program are significant. An action which is not lasting might be the creation of an object which is destroyed later in the execution of the program. The effects that remain after the execution of the program, that is, the differences between the state of the knowledge base before execution, and the state after execution, are known as the net side effects.

When IS-A is holds between two classes, any instance of the subclass must be an instance of the superclass. If the
actions involved in making an object an instance of the subclass include all the actions which would have been performed in making the object an instance of the superclass alone, the to-test and to-fetch programs of the superclass must surely recognize that object as an instance of that class. Thus if the set of net side effects of an add program for a class is a subset of the net side effects of another program, the latter program can be the add program for a subclass.

The net side effects of a program may include deletions of objects and assertions. If the net side effects of the to-kill program of a class form a subset of those of the to-kill program for a subclass, removing an object from the subclass will necessarily remove the object from the superclass. Thus restricting the programs in this way will insure that the subclass is indeed a valid specialization.

The test program of a class is not allowed to have net side effects. There is, however, a relationship between the values returned by such programs for IS-A related classes. In particular, for an object which is an instance of the subclass, the test programs of both classes must return true. If the object is not a member of the subclass, the test program for the subclass must return false, but that of the superclass may return either true or false. If the object is not a member of the superclass, it cannot be a member of the subclass. Hence the relationship between the values returned by the two programs is logical implication: the result returned by a test for membership of a subclass implies the result for the superclass.

Finally, one must consider the fetch programs of classes. Fetch programs are generators, programs whose executions suspend themselves to return a value and then may be reactivated to return another value. A fetch program would therefore return one instance of the class for each reactivation until the members of the class are exhausted and
the program terminates. When one considers the execution of a generator as the period from the first activation until the termination, such a program will have no net side effects. In addition, the value returned when the program terminates should be an indication to the calling program that termination has occurred, not a member of the class. Therefore, the relationship between fetch programs of IS-A related classes will not be characterized by restrictions on either the net side effects or the value returned.

The above discussion has indicated how a relation between programs must restrict their net side effects and values returned if this relation is to indicate when a program might be used as one of the four programs of a subclass of some class. In PSN programs are represented as classes and this relation between programs is IS-A. Therefore, if one program is to be a specialization of another, the structures of the two programs must be related as constrained by the inheritance rules described in chapter two. Inheritance, however, is not sufficient to insure that two IS-A related programs attached to IS-A related classes will act in the required manner. Therefore, for a program "p1" to be a legal specialization of "p2" it is necessary that when run in identical knowledge bases with the same parameters

1. If the programs return boolean values then the value returned by "p1" must logically imply the value returned by "p2". If the programs do not return boolean values, they must return identical values.

2. The set of net side effects of "p2" must be a subset of the net side effects of "p1". This means that the specialization must perform at least the changes done by "p2". "p1" is not a legal specialization if it performs the same actions as "p2" and then proceeds to perform other actions, some of which undo those performed in the first phase.

When "p2" is specialized to "p1", it is possible to restrict
the domain for which the program returns values; the above definition must hold only if the specialization executes successfully in a given state. Although the structural constraints imposed by IS-A on the specialization of programs is not sufficient to produce legal specializations, the mechanism does encourage the correct definition of IS-A related programs and makes it possible to specify guide lines for proper specialization.

III.3.2 The Structure of Programs

The basic declarative objects of a program are the parameters and variables, and the statements. If these behave like slots in a class in inheritance, one has immediately new tools for specializing programs. For example, the types of parameters will be the types of slots. Then to specialize a program one can specialize the types and possibly contribute more parameters. In a similar fashion, one might want to specialize the actions of individual statements, or add new statements which will add to the set of actions performed by the program.

In PSN all programs are classes, and parameters, variables, and statements are represented by slots. The various roles of slots are distinguished by the metaslots of which they are instances: parameters are instances of the metaslot "parameters". The metaslot tool is, in addition, used to add more structure to programs by dividing the evaluatable slots into three groups; the prerequisites are predicates, all of which must return true before program execution begins; the body containing statements which perform the actions of the program; and the returns statement which computes a final value to be returned.

Figure 3-4 illustrates the structure of the metaclass "PROGRAM" of which all programs must be instances. The metaslots of "PROGRAM" can be divided into two categories:
"parameters", "body", "prerequisites", and "returns" are those which the user uses to group his statements when writing the program, while the remainder are used to introduce the new properties which the slots of a program require. Thus any slot of the body of a program will have the properties "eval", "exception_action", and "exception" having types "FORM", "EXCEPTION_HANDLER_LINK", and "EXCEPTION_CLASSES" respectively. The IS-A hierarchy of metaslots within "PROGRAM" arises from the need to give the "eval" property to each of the "body", "prerequisites", and "returns" metaslots with slightly different definitions. Thus "eval" has the same meaning for both a "returns" slot and a "body" slot, but the property value associated with the "returns" slot must be an "EXPRESSION", while that associated with the "body" slot must be a "FORM".

The important innovation which makes programs executable is the "eval" property which "body", "prerequisites", and "returns" slots may now have. The type of the "eval" property is the class "FORM" or some subclass of this class. Instances of "FORM", known as forms, are used to invoke other programs. The forms used for "prerequisites" and "returns" slots must not have side effects. Instances of the classes "EXPRESSION" and "BOOLEAN_EXPRESSION" are forms which call programs without side effects. Additional restrictions on the statements of programs are that prerequisite slots must be of type "BOOLEAN" (the only instance of the class "BOOLEAN_CLASSES") and there must be exactly one "returns" slot (all programs return values).

As mentioned above, the value of the "eval" property is a form; this is a PSN object which can be interpreted to produce a value. When the structure of a form is not important, it is represented in diagrams by code in the PSN language enclosed in square brackets "[ ]". A particular example of a form is "[slot-name]" which means use the value of the slot with name "slot-name" when the form is interpreted. In the context of inheritance, it is important to note that forms too are classes and hence specialize through IS-A. Figure 3-5 shows how a
program to compute the factorial of its parameter might be represented. This program has one parameter, "n". The prerequisite "n_positive" checks that the value bound to "n" is greater than 0. If this is the case, execution proceeds with the interpretation of the form associated with the body slot "compute". When the value is returned, it is bound to "compute" in the same way that the actual parameter value is bound to "n". Thus the value to be returned is computed by finding the value bound to "compute".

Execution of a program involves creating an instance, known as a process, of the class and then computing values for all its slots. The value associated with the returns slot becomes the value to be returned. Initially, the form invoking the program supplies the values of the parameters properties of this process using a mechanism which will be described in the next section. The values for the slots of a process are the values of the returns slots of the processes created when the form associated with a slot is interpreted. The evaluation of the slots is divided into three groups: first the prerequisites, then the slots of the body, then the returns slot. The execution of the program will fail should any of the prerequisite slots be assigned the value false.

Within each of these groups, the order of execution is not predetermined; the mechanism for accessing the slots is through the "to-fetch" programs of the metaslots; thus to acquire all the prerequisites of a program "p1" the operation is to fetch all instances of the metaslot "prerequisites" which are PART-OF the class "p1". This lack of order has no major consequences when the slots being executed have no side effects as in the case of prerequisites. However, the consequences for statements in the body are rather severe: the side effects of any one statement must be independent of those of the other statements. As a concrete example, consider a case where one statement creates an object and another asserts that this object is related to some other object through some relation.
III.3 Programs

PROGRAM

factorial
parameters
prerequisites
n_positive
body
compute
returns
get_value

eval

n

type

NUMBER

type

BOOLEAN

eval

[n ≥ 0]

type

eval

[if n < 2
true 1
false
n*eval factorial
with n ← n-1
end if]

figure 3-5
Since the order of execution is not guaranteed, it is possible that the assertion would be attempted before the creation, a situation which must surely result in some sort of error.

When a program is specialized using IS-A, the programmer will be constrained by this lack of order to add only statements which are independent of those inherited. This implies that the new statements cannot act to undo the effects of the inherited statements, thus the new program must become a legal specialization. However, as will be explained later, the inherited statements may be modified in a way which will cause illegal specializations.

III.3.3 Forms

The objects which are the "eval" property values of the statement slots of a program are called forms. A form is defined recursively as

1. a reference to a slot;

2. a subclass of a program and an instance of the class "FORM" in which all parameters slots have either "eval" property values which are forms, or "quote" property values.

The relationship between forms and programs is like that of LISP; forms are expressions and programs correspond to lambda expressions (functions). The parameters slots of a program provide a means for binding the free variables in the forms invoked by the program.

This concept of forms provides, as it does for programs, a handle on the internal structure and a declarative meaning for specialization. It is here that the PSN description of programs differs most from that of TAXIS [Wong 1980]. The definition of programs is very similar to that of TAXIS transactions, but in TAXIS the objects corresponding to forms are expressions which are simply non-decomposable objects which
can be interpreted to produce effects and values. A TAXIS expression is always a specialization of another if certain conditions on the values returned and net side effects are satisfied. The lack of a declarative means of indicating specialization and the lack of internal structure make it impossible to define an expression as a set of modifications to something inherited from a more general expression. However, as with programs, an IS-A arc between PSN forms may represent specialization only if the conditions given in the previous section are satisfied.

Figure 3-6 shows the notations used to show the complete structure of forms. The program "SQUARE" computes the square of its parameter by invoking a form which calls the program "TIMES". The parameters of "TIMES" will both be bound to the value of the parameter "x" of "SQUARE". If a form is a slot reference, the arc labelled "eval" is drawn to the appropriate object. Other forms are simply drawn as classes. The form "FT" (representing "[times x,x]") demonstrates the advantages of making a form a subclass of the program: the parameters are passed to the form by the regular inheritance mechanism, and binding them to other forms involves only the addition of the "eval" values. Forms are, however, additionally distinguished from programs by having among their types the class "FORM".

This declarative mechanism allows forms to be specialized in two ways. The first is by using IS-A descendants of the forms to which the parameters are bound. The second is through the specialization of the program which the form calls. In this case it is necessary to explicitly make the new form an IS-A descendant of both the old form and the specialized program. Figure 3-7 shows the specialization of a form in which the specialized form, that associated with "a" in "P3", involves a call to a specialization of the program used in the IS-A parent.

When a form is executed, a corresponding process is
figure 3-6
III.3 Programs

figure 3-7

figure 3-8
created. This process is the instance of the program described in the last section. The first action is to acquire values for the parameters slots by evaluating the forms bound to the parameters; this evaluation may of course result in the creation of new processes. The dynamic sequence of the processes is recorded by assertions of the relation "dynamic" between them. The program instances which compute parameter values are joined by this relation to the process which caused the current process to be created. Once the parameters are all evaluated, the current program instance is linked via "dynamic" to the instance of the program which called the current program. "dynamic" is a PSM relation with interval restrictions which allow a processor to have no more than one successor and one predecessor.

Some parameters slots will be instances of the metaslot "quote_parameters", having "quote" links instead of "eval" links. In these cases the parameter assignment involves only the assertion that the parameter value is the value of the "quote" property of the slot. One use of this type of parameter is in a form which is a call to the identity function in which the single parameter is a quoted parameter. This produces the effect of the LISP QUOTE, as in the form "(QUOTE X)".

The "dynamic" relation is important in the evaluation of forms which reference slots; to evaluate a slot, the interpreter searches back along the "dynamic" arcs until a process is found which is an instance of a program of which the slot is a part. The value returned is then simply the corresponding slot value. This mechanism results in the usual interpretation of parameters and variables in recursion: the values found will be those most recently bound. Figure 3-8 shows a point in the evaluation of the form "[eval square with x<-2]" with the program "square" as illustrated in 3-6. Here the process which is an instance of the program "TIMES" has finished execution, but is still joined to the calling process
by "dynamic" and the returning value has not yet been linked. More specifically, the execution of the form "FT" will have begun with the creation of the unlabelled process. The "eval" property values of the slots "ax" and "ay" is the slot "x", therefore in parameter evaluation, the value assigned to each of the properties "ax" and "ay" is the value of the property "x" of the calling process, "p0111". Subsequently, the "eval" property of the "returns" slot, "prim" of "FT" is evaluated. This is a primitive function which computes the product of the two parameters and returns a value which then becomes the value of the "prim" property of the process. The next stage in interpretation involves asserting that "4" is the value of the "r" property of "p0111". Once the value has been returned in this way, the "dynamic" assertion will be removed and the instance of "FT" will be killed.

Since the variable bindings are associated with slot objects which are referenced in the structure of the programs, the scope of the variables is static (fixed at the time of creation of the program). A slot of the same name in two unrelated programs will be entirely different objects, and a form referencing one will never acquire the value from a process which is an instance of the program containing the other. One effect of this is that forms are tightly bound to the programs of which they are statements; an arbitrary form is not likely to be useful in more than one program because the slots which are referenced are not related.

Once the parameters are evaluated, execution proceeds as described in the previous section. A program called normally will execute to completion. The interpreter will then find the returned result and bind it to the appropriate slot of the calling process. At this point, the terminated process will be killed.

There is a second mechanism for invoking programs which provides a coroutine mechanism. A process may temporarily
suspend execution and return a value through a `detach` operation, which saves the state of the process, computes a value to be returned and places an instance of "DETACH" into the dynamic chain in the position of the original process. Thus the calling process acquires a value and kills a process as is normally done, but the original process remains and may be restarted by any other process knowing the identity of the detached object. When such a process returns a new object in a class or a relation each time it is invoked until all such objects are exhausted, it is known as a `generator`.

### III.3.4 Sequential Program Constructs

A previous section has described the non-deterministic evaluation of the statements of a program; it was pointed out that this forces a programmer to specialize programs in the desired way. It is, however, not clear that all desirable programs may be written with this mechanism. In any case, it is possible, using only the mechanisms illustrated so far, to create a sequence of forms which must be executed sequentially.

Figure 3-9 illustrates a program of two arguments known as "BEGIN". Its first parameter may be any object, the second must be a form. If the second parameter is identical to an object known as "NULL_FORM", the program will return the value of the first parameter, otherwise it will return the result of evaluating the second parameter. "NULL_FORM" is a form which is a subclass of "NULL_PROGRAM". This is a class entirely without structure, hence evaluation in any domain will have no net side effects, and the returned value will always be unknown. Therefore, any program may be a specialization of "NULL_PROGRAM" and, hence, any form a specialization of "NULL_FORM".

Figure 3-10 shows a chain of forms which are subclasses of "BEGIN" joined together by "quote" links. The "arg1" parameters have as their "eval" properties another set of
III.3 Programs

BEGIN

parameters
arg1
arg2
returns
r

type
OBJECT
FORM

eval
[if arg2 = NULL_FORM
true arg1
false interpret arg2
end if]

figure 3-9

BEGIN

parameters
arg1
arg2
returns

F1
eval
quote

parameters
arg1
arg2
eval
quote

parameters
arg1
arg2
returns

NULL_FORM
quote

F2
eval
quote

F3
eval
quote

figure 3-10
forms. Until the last "BEGIN" form is reached, the sequence of operations will be (for each "BEGIN"):  

1. create the process and associate the parameters properties; the "arg1" property value will be found by evaluating the form which is its "eval" property; the "arg2" property value is simply the "quote" property  

2. the "arg2" property value will be interpreted.

This mechanism then, allows the guaranteed sequential evaluation of the forms associated with the "arg1" properties of the "BEGIN" forms. Thus, in the example of figure 3-10, the form "F1" will always be interpreted before "F2", and "F3" will be executed last.

This sequence of forms has another interesting property. Since any form may be a valid specialization of "NULL_FORM", a specialization of the last "BEGIN" form in the sequence may be made by replacing the "quote" property by another "BEGIN" form. This specialization can then be made the "quote" property value of "arg2" in the previous form. The process may be continued up to a program definition as shown in figure 3-11. What has resulted here is a specialization of program by the addition of a form at the end of a group of forms which are sequentially executed. This structure of "BEGIN" forms is the means by which a "begin - end" block may be implemented in PSN.

This structure, of course, makes it possible once again, to create IS-A children of a program which are not legal specializations. For example, when creating the program "P2" in figure 3-11, the programmer could specify an "P3" which would undo a side effect caused by "F1" or "P2". It is the responsibility of the programmer to avoid doing so.

In summary, the rules for producing legal specializations of programs are

1. make the actions associated with each slot independent
III.3 Programs

figure 3-11
of those associated with the other slots, and

2. avoid the addition of forms at the end of begin blocks which cancel the actions of inherited forms.

III.4. Exceptions

At times in the execution of a PSN command one finds that execution can not continue because some constraint or prerequisite is not satisfied. Such an occurrence is known as an exception. There are two types of exceptions which can arise in the PSN formalism. One type arises from an inconsistency in the state of the knowledge base; for example, an object having a slot value which violates a restriction on the slot would be exceptional. Exceptional objects might be allowed in the knowledge base if their existence is explained by an object known as an excuse. The handling of such exceptions is discussed in detail in [Lesperance 1980].

The second type of exception is the dynamic exception in the execution of programs. In programming languages an exception arises when some illegal action, such as division by zero, occurs during execution. The usual action is to set a flag describing the exception, aborting the routine in which the exception occurred, and activating a special block of code, known as an exception handler, in the calling routine. The exception handler will compute a value which is to be returned as the value for the routine which failed. For example, a routine for division will usually fail when the divisor is zero; in this case the division is aborted and a flag for a "zero-divide" exception is set. This is known as raising the exception. The calling routine may contain an exception handler for this exception which might decide that the value of the division should be zero. A different calling routine might contain a different handler, one that decides that the value should be that of the largest representable number. The
III.4 Exceptions

exception handling mechanism is thus flexible in that the actual response to the exception depends on the context in which it occurs.

The specific details of this exception handling mechanism in PSN are taken from TAXIS [Wong 1980]. Modifications were necessary in areas where the two formalisms do not match exactly and to adapt the mechanism to the PSN definitions of inheritance. In PSN an exception occurs in the execution of a program when the evaluation of the "eval" property of any of the prerequisite, body, or returns slots fails or if the value returned to a prerequisite slot is false. Associated with each of these slots is an "exception" property value which is a form which creates an instance of the metaclass "EXCEPTION_CLASSES". When the evaluation fails, the process of raising the exception involves executing this form to create an instance of the exception class. The TAXIS mechanism allows exceptions to be raised only when the value returned to a prerequisite is false. PSN however allows a statement called "fail" to be executed, causing a form to fail. Also, an exception will be raised for a slot if it is incapable of handling an exception raised in its evaluation.

As described above, the action to be performed to handle the exception is specified in the calling program. More specifically, any evaluable slot has a value for the "exception-action" slot which is of type "EXCEPTION-HANDLER-LINK". An instance of this metaclass has slots whose types are exception classes (instances of "EXCEPTION_CLASSES") and have in addition associated programs via the "handler" property. Figure 3-12 shows an example of the various items associated with exceptions. When an exception is raised, the interpreter finds the calling process by following the dynamic arc; it can then determine the active slot and create an instance of the corresponding "exception-action" value. It then searches for a slot in the exception handler link which has as type a class which is a
III.4 Exceptions

figure 3-12
type of the exception which was raised. The program which is the "handler" for this slot is then executed with the exception as a parameter, and the returned value is used for the value of the slot whose form originally failed.

This mechanism differs in one important way from that of Taxis. In Taxis, control is returned to the next expression in the program in which the exception was raised, while PSW replaces this program with the exception handler. It is difficult to see how the zero divide problem would be fixed using the Taxis mechanisms.

As an example, consider again the objects in figure 3-12. If the division in "PROGRAM-2" should fail perhaps in an attempt to divide by zero, an instance of the exception "CANNOT-DIVIDE" is raised. The dynamic link is then followed back to the instance of "PROGRAM-1" where "EHL1" is the exception handler link for the active slot. In this class, the slot "b" has type "CANNOT-DIVIDE", thus the associated handler, "PROGRAM-4" is executed, and the value returned is bound to the slot "a2" in "PROGRAM-2" and execution continues.

Since exceptions and exception handler links are classes, the IS-A constraint on "exception", "exception-action", and "handler" links is simply the requirement that the values remain unchanged or that the values for the specialized slots are IS-A those of the originals. This gives the exception mechanism the full flexibility provided to programs by the inheritance mechanism. Thus when a program is specialized, the exceptions associated with various slots might be specialized, the handlers for various exceptions may be specialized, and more kinds of exceptions may be provided with handlers. Figure 3-13 illustrates specialization of handler links, and figure 3-14 shows the specialization of exception classes.

There are of course default actions which occur when no exception class or exception handler links are specified. The
III.4 Exceptions

figure 3-13

figure 3-14
default exception for slots is the class "GENERAL_EXCEPTION". All exception classes must be subclasses of this class so that the default handler link can deal with the exception. The default handler link joins the "GENERAL_EXCEPTION" to the program "abort"; therefore if any exception is raised and no handler is specified, the program will be aborted. Handler links need not be subclasses of this general link, and it is thus possible that no handler is supplied for an exception. In such cases the interpreter will abort the operation.

Exception handlers always receive the exception they are to deal with as a parameter value for the slot "exception". This requires that all programs which are used to handle exceptions be subclasses of the program "GENERAL_HANDLER" which simply supplies the definition for this parameter.
CHAPTER IV
Implementation - Overview and Primitives

IV.1. Overview

The previous two chapters have described PSN as a semantic network consisting of objects and relations between them. Diagrams were used to illustrate parts of such a network. The formalism also involves the ability to affect changes in this static representation; the forms and programs introduced in chapter three are structures in the network which can be interpreted to cause transformations. The emphasis in an implementation is on the introduction of such forms and programs into the knowledge base and on their subsequent execution.

The representation of the network cannot, however, be ignored. For programs to be useful there must be objects and relationships on which they can operate. One of the concerns of this chapter is the description of a storage representation of a semantic network for the PSN system. This representation is a distinct level of the implementation in that the interface with the remaining system is through a small number of operations which may easily be redefined should a different representation be desired.

The opposite pole of the implementation is the translation of the PSN language into objects, especially forms, in the knowledge base. For example, if the user types

"make John instanceof PERSON",

the translator creates a form which is basically a call to the to-add program of "PERSON" with "John" as the value of the parameter representing the object which should be made an
instance of the class. A form more commonly found in other computer systems might be an arithmetic operation such as "4 + 5"

which would cause a call to the "+" program. More aspects of this language will be discussed in chapter five, and a complete syntax is provided in the appendix.

Once the form entered by a user is compiled, the new object in the knowledge base is interpreted. The interpreter routine is a major component of the system and is also described in chapter five. The result of the interpretation will be a PSN object (possibly "unknown"). The external name of this object will be printed for the user; in the case of the addition, for example, "9" would be printed at the terminal.

The creation of objects will be a very frequent occurrence in the use of PSN. In particular, each command which is entered results in the creation of at least one class. Therefore the system offers as primitive routines for creating objects, applying the add programs of classes, adding IS-A relationships, and adding and inheriting structure. A routine is primitive if it appears as a black box to PSN (i.e., its actions are not reflected in its internal structure). Primitive routines are generally the values of "eval" properties of "returns" slots of system supplied PSN programs. This chapter looks at the algorithms used in some primitive routines while chapter five will discuss the system supplied programs. It is possible to provide a PSN system where the only primitive routines are those which interface with the stored knowledge base, but routines encoded in LISP execute far more efficiently than PSN programs.

The final aspect of the system is a set of programs which are the default programs for user created classes and relations. The standard routines are also coded as primitive routines. These routines are what will generally be used (with possible additional actions) to provide the semantics of stored
classes and relations, that is, those classes and relations which acknowledge instances by a specific change to the knowledge base. For a given class it is not required that the user use standard routines; for example, he could create a relation called "my_instance_of" and add objects to classes by asserting this relation. The relation would however also need four programs, and if these are not to be the standard programs, additional classes or relations are required to store the assertion. It is clear, therefore, that at some time a primitive function will be needed to store an assertion or an instance. A discussion of the standard routines forms the concluding sections of this chapter.

IV.2. Storage Representation of PSN Objects

The representation of the semantic network has two levels: the first is the actual physical storage which for this system uses LISP structures (but could perhaps use records on a disk, etc.), and the second is the use of this storage in representing objects, slots, assertions, IS-A, etc. The interface to the storage level is very simple: the standard parameters involve the name of a PSN object and a storage field name (property). In retrieving from storage these parameters are sufficient to specify a value which is usually the name of a PSN object. When a value is to be stored the value must additionally be specified. Certain classes of objects are, however, not represented in the standard fashion. Such objects are strings, numbers, and intervals and are characterized by the fact that they behave as if all such objects permanently exist (they cannot be created nor destroyed) and that they may have no internal structure.

Subsection two and following describe the higher levels involved in representing a semantic network. The problems solved there are representing classes and slots, storing IS-A between classes, representing assertions, and associating property values with objects. Of particular importance is the
IV.2 Object Storage

association of slots with classes. In PSN, although an inherited slot may have different properties than it has in a superclass, a slot is a unique object irrespective of the number of classes in which it appears. It was necessary, therefore to devise a mechanism by which the appropriate view of a slot is associated with a class. In this representation inheritance of structure is completed at the time of creation of the slot; that is, it is not necessary to search via IS-A for the appropriate view of a slot. This mechanism does, however restrict the definition of structure and the assertion of IS-A to the time of creation of a class.

IV.2.1 Standard Objects

Most PSN objects are stored as unique LISP atoms; the names of these atoms are the internal names of objects and play the same role with respect to the implemented net as do external names to the formal net. The property lists of these atoms are used to store the relationships that the atom has in a semantic network. All atoms have a "SYSTEM:" property which stores the external name of the object, and an "INSTANCEOF:" property which stores a list of the types of the object.

Although most of the system defined properties of an object are stored as values on the LISP property lists, the higher levels of the system do not depend on this representation. All access to properties of objects are made through a set of functions which decide how to store or fetch a value given an object and a property, hence in later extensions of the system it will be possible to change the representation of certain classes of objects or properties without modification of the majority of the system.

The terms property and property value now have three different meanings in various contexts. When discussing properties in the formal view of PSN, a property of an object corresponds to a property definition or slot in one of its
types, and a property value is a slot value, a value associated to the object by that property. In the general discussion of the implementation a property is a LISP atom which is used with atoms representing PSN objects to access s-expressions stored with the objects via the basic functions. In this context the property "INSTANCE-OF:" is an atom which when passed to the basic retrieval function with some object will reference the list of classes of which the object is an instance. Finally, the third view is the standard LISP view in which a property is an atomic flag which when occurring in a property list of an atom indicates a value. In the remaining discussion, the last of these meanings will not be intended unless it is specifically stated. Which of the other two meanings is intended will be obvious from the context of its use.

Certain classes of objects behave as if every instance permanently exists in the knowledge base; that is, any such object can be referenced from the input language and be used without previously being created, unreferenced objects from these classes are not explicitly stored, and references to these objects are not uniquely stored. Such objects are instances of classes which P. Schneider [Schneider 1978a] has called intensional, and examples are numbers and strings. In general, these objects have certain restrictions on their use: they have no internal structure and no properties may be assigned to them; they may not appear as domain instances of assertions; the range interval of assertions in which they appear is not checked; they may not be created or destroyed; and, relations which have such classes as ranges are not invertable by the standard routines. Other such objects are type sets consisting of a list of classes, intervals, and slot-specs, a special type of object used by the system for referencing slots.

These last three types of objects are treated as classes in some sense, for although they may not have instances, the objects all participate in an IS-A hierarchy. These
hierarchies are predefined and permanent — it is impossible for the user to modify them. For type sets, "A" IS-A "B" if and only if for each element of "B" there is an element of "A" which is the same as or an IS-A descendant of the element of "B". This is equivalent to saying that if an object "x" is an instance of every class in "A" then it is an instance of every class in "B", hence type sets are used as the values of the "type" properties of slots or the set of classes of which an object is an instance. Type sets are stored as non-redundant type sets in which none of the classes of the set is an IS-A descendant of or is equal to any other member of the set.

The predefined IS-A for intervals is simply containment: if "]i, j]" IS-A "]k, l]" then i is greater than or equal to k and j is less than or equal to l. Of course, all valid intervals must have the lower bound less than or equal to the upper bound.

A slot specification is an object which indicates a reference to a slot. Later, it will be seen that redefinitions of a slot, although formally the same object, are stored as distinct internal objects. A slot specification consists of a class object and an external slot name and is used by the interpreter to resolve variable bindings. This operation involves finding a process which contains among its types a class which has as a part the given slot. Given a slot specification, the search becomes a search for a process which contains among its types the class of a slot specification or an IS-A descendant of this class. When this class is found, the internal representation of the appropriate version of the slot may be found. One slot specification is a specialization of another if the class of the first is an IS-A descendant of the class of the first, and the slot names are identical.
Current discussions in PSM rely heavily on the concepts of classes and the IS-A hierarchy. Therefore, this implementation provides mechanisms which implement the special features of classes as primitive operations. In particular, an atom representing a class may have any of five special system properties: "INSTANCES:“, "ISA:“, "ISA-CHILDREN:“, "STRUCTURE:“, and "META-STRUCTURE:“. Classes which use the standard to-add program will keep track of their instances by maintaining a list of instances as the value of the "INSTANCES:" property. "ISA:" and "ISA-CHILDREN:" implement the IS-A hierarchy, the value of the "ISA:" property being a non-redundant set of classes of which a class is an IS-A descendant, and the "ISA-CHILDREN:" property being a list of all classes which include the class as an element of their "ISA:" properties. "STRUCTURE:" and "META-STRUCTURE:" implement the PART-OF relation where each stores a list of associations of external names to internal objects, the instances of "slot" being stored as structure, and instances of metaslot being stored as metastructure.

Every class stores this complete list of internal objects so that the slot accessing mechanism need not concern itself with IS-A. However, objects which are not metaslots and are inherited without modification are stored uniquely; if "A" IS-A "B" and "a" is a slot inherited from "B" with no changes, then references to "a" in the context of either "A" or "B" will produce the same internal object. On the other hand, modification in inheritance will result in a new internal object. This is one means of implementing a limited context mechanism: externally a slot is to be viewed as a unique object whose properties are to be accessed in the context of the class being referenced. The various internal representations of a slot or metaslot are bound together in a redefinition hierarchy implemented by the system properties "REDEFINES:" and "REDEFINED-BY:" whose values are related in a fashion similar
to that of IS-A. In particular a non-redundant set of internal representations of a slot (or metaslot) is a set in which no object is a redefinition of another.

The "SYSTEM:" property of a slot is a LISP dotted pair of which the CAR is the metaslot of which the top of the redefinition hierarchy is an instance (at present a slot may be an instance of only one metaslot), and the CDR is the external name of the slot.

The PART-OF relation, when implemented this way, is not easily invertable, that is, stored links reference slot representations from classes, but no stored links reference defining classes from the slots.

IV.2.3 Storing and Accessing Property Values

The association between objects and their property values are stored as properties on the LISP property list of the object where the name used is the atom which is the internal representation of the slot which defines the particular relationship. Thus the operations of storing and retrieving slot values must involve the class of which the object is an instance so that the appropriate internal object may be found. The operation "IDENTIFY-SLOT" takes as parameters a list of classes and an external slot name and searches the structures of the classes for slots having that name. Should more than one internal object be returned, the non-redundant set of objects is computed. If this set contains a single object, the identification succeeds, otherwise if fails.

When a slot value reference is made with respect to some object, the set of classes passed to the identification operation is the non-redundant set of types of the object. The object which is then returned can be used for storing or retrieving values. The storage operation will check that the proposed value satisfies the types and restrictions of the
IV.2 Object Storage

slot, hence the identification of the internal object was necessary even had some other storage mechanism been implemented.

Value retrieval also requires the identity of the internal representation to access properties of the slot; specifically, the value of the default property is often required. The operation sequence first attempts to fetch a value. If none is stored and the object is a class, an attempt to find an inherited value is made. In inheritance, the algorithm is to find the set of all values for the appropriate slot for all IS-A parents of the class which have the property. If this set has more than one member all of which are classes, then compute the non-redundant set of classes. In any case, if a unique value is found, then this is the inherited value, otherwise, nothing is inherited and the default value, if any, is used. When there is no default value at this point, the special object "unknown" is returned as the value of the retrieval.

IV.2.4 Storing Relation Instances

The standard program for storing assertions of PSN relations stores the assertion by adding properties to both the domain and range objects. These properties are complex in that the retrieval routine requires more than just a property and an object. The first such property is the "D-R-LIST:" which is associated with a domain instance of an assertion. The retrieval operation requires the property, the domain instance, and the relation object and returns an s-expression of the form

"[(relation)<count><inst1><inst2> ...]"

where count is the number of elements following and the "<insti>" are objects which are the range instances of the assertions of the relation in which the domain instance is involved. The property "R-D-LIST:" is the opposite; it references the list of domain instances for a range instance. A particular assertion is then represented by a pointer to the range instance in the "D-R-LIST:" property of the domain
instance and a pointer to the domain instance from the range instance in the "R-D-LIST:" property of the range instance. The latter will be missing in the case of relations which are not invertable because their domains are intensional classes.

The retrieval routines are also capable of retrieving the number of assertions an object is involved in as a domain instance given the "DOMAIN-INTERVAL-COUNT:" property, the object, and the relation. The "RANGE-INTERVAL-COUNT:" property similarly results in the number of assertions the object participates in as a range instance.

IV.3. Creating objects

As mentioned in the introduction to this chapter, the algorithms for creating objects are encoded as primitive operations. The routine which is most often used for this purpose includes a number of other primitive operations. Firstly, whenever an object is created, it must become an instance of some class. All objects are at least instances of the class "OBJECT". Thus when an object is created, the to-add program of at least one class is applied to it. The second restriction on the creation of objects is the fact that a class must be given its structure and IS-A parents at this time. The routine will then invoke the algorithms for adding and inheriting structure.

IV.3.1 Objects

The routine for creating objects takes a number of parameters, of which all but the list of types are optional. The other parameters that concern the user are a list of IS-A parents (for classes), a data structure which describes the structure of a class, and a list associating properties to slot values. Most of the operations shown here can be invoked separately although almost invariably the user will cause them to be run as calls from "NEW-FORM-BODY", a primitive routine
The first action is to create a LISP atom as the storage location of the new object. Usually this involves using GENSYM to create a new atom. Often, however, for internal reasons, the system will have created this atom earlier and pass this atom as a parameter to "NEW-FORM-BODY". The user may never himself supply a name for this slot because he has no mechanism for generating a new location and will therefore make the system inconsistent. The actual creation is done by a basic function called "CREATE-OBJECT", hence apart from the internal name, the system need know nothing about the representation of objects.

With this newly created object, the routine now associates the types to the object by the "INSTANCEOF:" property, storing as mentioned before, only the non-redundant type set. This does not make the object an instance of its types; this property is necessary only for the slot value attachment mechanism. However, the system does make use of the types to determine whether the new object is a class.

If the object is to be a class, it is necessary to assert the "ISA:" property and link the class to its IS-A parents as described in the subsection on the storage of classes. All classes are in addition made IS-A descendants of "OBJECT" and all programs are made IS-A descendants of "NULL_PROGRAM" and forms of "NULL_FORM".

The second stage for classes is to add the structure, using that inherited from IS-A parents and that supplied as a parameter value. The details of this are described in the next subsection. The special actions performed for classes can be considered actions which should be performed by the "to-add" program of "CLASS" which are taken inside the interpreter because of their common application to every instance of "CLASS".
IV.3 Creating

The final step is to run the "to-add" program for each of the types of the object so that the object actually becomes an instance of these classes. Here it is important that only the add programs of classes in a non-redundant type set are used, for were the add programs of both a class and some subclass of that class to be run, the common actions would be duplicated resulting in possible inconsistencies in the knowledge base.

IV.3.2 Adding Structure

The routine for adding structures has as parameters a list structure defining the new features of a structure, an inherited structure, and a metastructure. The inherited structure will in general arise from more than one IS-A parent, and the metastructure will arise from more than one type. The mapping from many structures to one structure is done by a merging operation which is applied to the total structure to produce the inherited structure, and to only metastructures to produce the metastructure.

The merging operation consists simply of checking that there exists a unique redefinition among the set of inherited views of each slot. Since slots are referenced by their external names by the system, two unrelated slots with the same name will result in a set of slots for which there is no unique redefinition. Therefore, in this implementation, it is not legal to inherit different slots with non-unique names. It is also not legal to inherit a set of different redefinitions of a slot for which there is no unique redefinition.

For each inherited slot, the routine checks for modifications desired in the properties of the slot. If such do not exist and the slot is not a metaslot, no action is taken. Otherwise, the operation of redefining the slot proceeds; the term old slot refers to the inherited slot and new slot to the redefinition which is being created. There are two prerequisites to the redefinition operation:
1. among the types of the new slot (the non-redundant sub-type set of the union of the types of the old slot and the newly specified types) there must be a class which is a member of the merged metastructure;

2. there may be only one metaslot in the non-redundant type set of the new slot.

If these prerequisites are satisfied the operations are:

1. An object is created using the routine of the previous section, exactly as a normal object with the exception that no property values are passed to the add routine (the slot redefinition routine will assign them in the next step). A result of this is that the fetch program for "slot" will fetch as distinct objects the various redefinitions of a slot.

2. The property values of the new slot are assigned: for most properties, the new value is compared with the old value (if any) to check that the IS-A constraint is satisfied; in the case of the "type" and "restriction" properties, a new value is computed from the old value and the value supplied as a parameter. The "type" property of the new slot is computed as the non-redundant type set contained in the union of the value for the old slot and the newly supplied types, hence the IS-A constraint is automatically satisfied. Similarly, the restriction for the new slot is a form which is the logical "and" of the old restriction and any newly supplied restriction.

3. The "REDEFINES:“, "REDEFINED-BY:“, and "SYSTEM:“ properties are asserted for the appropriate objects.

When these operations have been carried out for all inherited slots, there may remain unused parts of the structure definition parameter. These are taken as definitions of new
IV.3 Creating slots whose creation simply involves a call to the object creation routine followed by the assertion of the appropriate "SYSTEM:" property. In this case, the property values are passed, therefore the creation of new slots is exactly the creation of new objects.

IV.4. The Standard Routines - Adding

IV.4.1 Objects

The standard routine for adding objects to a class simply causes the object to be stored in the list of objects which is the value of the "INSTANCES:" property of the class. The routine will not accept instances of intensional classes such as "NUMBER".

IV.4.2 Assertions

The standard routine for adding assertions of a relation between two objects has a number of prerequisites:

1. The source of the proposed assertion must be an instance of the domain of the relation, and, similarly, the target of the assertion must be an instance of the range.

2. An assertion of the same relation must not already hold between the two objects. In addition, there must be no assertion with this source and target of any relation which is an IS-A ancestor of the relation, because the new assertion would result in two instances of this ancestor.

3. The proposed assertion must not violate the "d-int" (domain interval) of the relation or any IS-A ancestor for the source object. Similarly, the target may not participate as the target in more assertions than are allowed by the "r-int" property of the relation and its ancestors.
The algorithm for the last two conditions involve a search of the IS-A hierarchy. For condition two, the test program of the relation and each ancestor must be tried on the source and target. For condition three, for each of the same relations, a fetch must be performed for all assertions in which the source object is the source. These objects must then be counted and the "r-int" checked. The algorithm avoids much work by only performing the fetch if the upper bound is finite. The lower bound need not be checked because the fetch programs should return at least the minimum number of objects, filling in "unknown" to meet this number. Also, the program assumes that all relations in the IS-A hierarchy in which the current relation resides will use the same assert program. This allows the fetching operations to be written into this program in such a way so that the instances for each relation are fetched only once, even if the IS-A hierarchy is not a tree.

Once the prerequisites are satisfied, the assertion is stored as described in section IV.2.4.

IV.5. Standard Routines - Recognizers

IV.5.1 Objects

The standard test routine must return "true" if the instance which it is recognizing occurs in the given class or any descendant of this class. Because the standard INSTANCE-OF assertion is stored with only the particular class of which the object is a direct instance, it will often be necessary that this program search down the IS-A hierarchy to verify membership. The program simply calls a recursive program which returns "false" if the class (its parameter) has been marked, returns "true" if the object is stored directly under the present class, and otherwise marks the class as checked and calls itself with each of the subclasses of the present class. The marking is necessary, of course, because the IS-A hierarchy need not be a tree, hence a class may be reached in more than
one way as a descendant of some other class.

IV.5.2 Assertions

This operation is very similar to that for non-assertion objects because each assertion is found through the relation of which it is a direct instance. Again a search of the IS-A hierarchy in which each relation is marked as it is checked is necessary. The existence of the assertion must be checked only in one direction.


The operation of killing is simply the inverse of the addition operation. Fetching corresponds to testing because it also involves searching the IS-A hierarchy to find all instances. These algorithms do not require further discussion.
V.1. Introduction

A major shortcoming of previous implementations of PSN has been the lack of an input language designed to express the concepts of the formalism. These implementations were simply extensions to LISP in which the standard programs for classes and relations and special programs for such objects as "CLASS" and "RELATION" were supplied. In general, the list structures required as parameters were complicated and tedious to use.

This problem is magnified when one tries to create program objects as described in chapter three. Programs usually require that a number of form objects be created, a task which in older systems would require a number of calls to the creation routines in which the most nested forms (for example, the last statement of a begin block) must be created first. The possible solutions are to either write LISP functions for various specialized creations which can have simplified calling sequences, or to provide a parser for a new input language. Section two of this chapter describes some features of such a language and some aspects of the translator for this language.

Section three describes an interpreter which can execute the objects which describe programs. This includes a discussion of how primitive operations are recognized, and the implementation of the relation "dynamic". The final section describes the implementation of several predefined programs and their resulting forms; these are the forms generated by the parser to implement the various operations such as creating objects, begin blocks, for loops, etc.
V.2 Parsing

V.2. Parsing

V.2.1 The language

The PSN language used in this implementation is based on the language introduced in [Levesque 1977]. Many of the features and shorthands in Levesque's language have not been included. The design goals were

1. to provide the basic mechanisms of PSN, that is, the capability to specify procedural attachment, create objects and assertions, and assert that objects have types;

2. to provide basic programming language constructs with which programs specifying the semantics of PSN objects could be written;

3. to provide a language that can be parsed deterministically with one or two token look ahead;

4. to provide some the declarative aids, such as IS-A, which simplify the task of building knowledge bases.

PSN differs from conventional programming languages in that all operations concern themselves with objects in the semantic network, while conventional languages concern themselves with simpler data structures. The names which appear in PSN are not variables but external names for objects; these names are constants and cannot take on new values unless the object referred to is destroyed. There are, however, the standard static scoping rules: a name may be given a different meaning inside a begin block or program definition.

In a PSN program slots are used as variables. Variable bindings are implemented by the association of the appropriate value to an instance of the program through the property defined by the slot. When writing programs in the language,
the use of the slot as a variable is indicated by using the external name of the slot. For example

```
"factorial := program
  parameters
    n := a NUMBER end slot
  prerequisites
    pr1 := a BOOLEAN with
    eval <- [n >= 0]
... end program"
```

shows the slot referred to by "n" being used as a variable. It is important to recall that once such a variable has been assigned a value, the assignment cannot be changed.

The other important aspects of the language involve the invocation of the four programs associated with classes and relations. For example,

```
"make Helen instanceof STUDENT"
```

invokes the add program of the class "STUDENT". A unique feature is the for loop which takes an index variable through the values returned by a generator as in

```
"for x in PERSON do print x end"
```

In this case the generator is the to-fetch program of the class "PERSON". Section four of this chapter explains how this can be implemented in PSN.

Most of the language constructs begin and end with a reserved keyword so that parsing is very simple. For example, the examples above show that a program definition begins with "program" and ends with "end program". Options are signalled by a keyword beginning the option before the keyword ending the construct. Thus in creating a new object one can specify IS-A parents, structure, and slot values as in

```
"new CLASS isa ... structure ... with ... end new"
```
V.2 The Scanner

The purpose of the scanner is to provide the parser with a sequence of tokens. Parsing is interactive: after a line of text is read the parser works until the tokens supplied are exhausted. This arrangement allows the introduction of interactive control of the process of parsing. This subsection explains how the interactive control has been included. The scanner used for the system is basically an implementation of an algorithm discussed in [Gries 1971].

When the parser requires a new token it calls a routine called "ADVANCE" which pops a token off the list of input tokens. These tokens are either LISP atoms representing keywords and printable operators, or pairs

"(<token type> . <literal>)"

where the token type is one of "NAME", "STRING", "NUMBER", or the name of an operator, and the literal is the print version of the token. "ADVANCE" sets the token type and literal for the parser, using the token type as the literal when there is no literal.

When the list of input tokens is exhausted, "ADVANCE" calls a routine to produce a new list of tokens from the input buffer. This routine, which is known as the scanner, processes an entire line of input at a time. Its control loop works as follows: if the last token found was a control token, perform the required action, otherwise, add this token to the list of tokens to be returned and invoke the scan routine associated with the current character in the input buffer and assign the result to the last token found. Initially, the last token found is set so that the program runs the scan program for the first token in the buffer.

The control tokens are:

1. end of line: This results when a carriage return, a line feed, an or escape character is typed and indicates that
the end of line has been reached. The scanner returns the list of tokens which it has constructed.

2. break: When a control-B is detected, the RUTGERS/UCI LISP break package is entered in which the user has full use of LISP. The current token will be set to the value returned from the break.

3. abort: A control-A will cause the current parse to be aborted.

4. delete: Control-D causes the deletion of the previous token.

5. edit: When control-E is typed, the user will be able to edit his input list after the next end of line.

6. fill: The character control-N indicates that the next token is the name of a variable referencing a list of valid tokens. This list is appended to the list that the scanner is building, and the combined list is returned to "ADVANCE". This action must be the last on a line.

Associated with each of the 128 ASCII characters, in addition to the scan programs, is a type. These types are used by the scan programs; for example the letters are of type "LETTER". Characters which appear only as single character tokens have as their types their token representation, and control characters have as their types LISP atoms which have a "CONTROL:" property on their property lists. The scan program for single character tokens simply returns the type of that character. The program for a letter will read letters until a token whose type is not "LETTER" or "NUMBER", creating a LISP atom from these characters. If such an atom has a "KEYWORD:" property, the atomic token for that keyword is returned. Otherwise, the token "(NAME . <atom>)" is returned. For characters which begin other composite tokens (for example,
"<-") the type is necessary only to indicate to the programs for scanning names and numbers that the character does not belong to such tokens.

V.2.3 Parsing

The translator operates in two stages. The first stage (the parser) produces an intermediate structure which guides the second stage in creating the objects representing the input. The major concerns of the parser are the resolution of external names, interactive error correction, and creating forms for primitive operations. The second stage simply calls the relevant primitive routines.

The output of the parser is a LISP s-expression which may be used as a parameter to a routine which can create a form in the knowledge base. This allows changes to the parser without requiring changes to the rest of the system. Some communication through the internal representation of objects is, however, required. Firstly, the parser maintains a list of external names of an object as a value on the property list of the LISP atom representing it. Secondly, to resolve references to uncreated objects, the parser may pass atoms to the creation routine which are to be used as locations for storing the objects which are to be created. For example, when a statement in a program references a parameter, a slot specification must be generated containing the internal name of the class. When the parser generates a name for that class (using GENSYM), it can pass the completed slot specification along with the instructions for creating the entire class.

Parsing is accomplished through a simple recursive descent algorithm (as described in [Gries 1971]). Each rule of the grammar is represented by a LISP function which possibly calls other rules and produces an s-expression which the calling routine may use to help construct the result of the parse. Most of these functions have no parameters; the exception is
that for slots which requires as a parameter the meta-slot of which the slot is to be an instance (how meta-slots are specified is discussed in GRAMMAR). Each rule expects an unused current token on entry, and is expected to leave an unused token when it returns.

Interactive error correction is possible because each function is required to store the current status on a parse stack. If an error occurs, the user can restore the status at some higher point in the parse tree and re-start execution at that function after editing the input list. An error handling routine is provided which has commands for

1. changing the state to some previous non-terminal,
2. inserting and deleting tokens from the input list at that point, and
3. causing execution to restart at that point.

The last feature depends heavily on the RUTGERS/UCI LISP evaluation mechanism and is not portable.

The parser also includes some functions which take as arguments structures returned by the functions for non-terminals of the grammar and rearrange or add to these structures. The most important of these is an LEXPR called "FORM". This takes as arguments the name of a primitive function (for example, ADD) and an arbitrary number of s-expressions representing parsed expressions which are to be arguments of the primitive function. From these an s-expression is made which will result in a form whose "eval" properties are the forms resulting from the argument expressions and which has as a "returns" statement a call to the primitive operation. The next section will describe how the interpreter handles such forms.
V.3 Interpreting

V.3. Interpreting

V.3.1 Introduction

The interpreter is the routine which deciphers the structure of PSN programs and forms to cause changes in the knowledge base. It therefore implements programs as they were described in chapter three. The important issues are the implementation of the relation "dynamic", the mechanism for returning values, the means by which a generator can be created and detached (returning a value), parameter evaluation, variable binding, and execution of forms which are the "eval" property values of slots.

At some level, the "eval" links will point to primitive forms which are black boxes to PSN. The system represents these forms as LISP s-expressions which are simply given to EVAL in LISP. These LISP s-expressions are recognized as instances of the class "FORM", thus they may be validly used as "eval" property values. The input language, however, does not give the user the ability to directly create such forms. Therefore, they appear only where generated by the parser to implement expressions and in the special predefined programs which will be discussed in the following sections.

The LISP forms generally acquire their actual parameters from the PSN forms which invoke them. The mechanism for doing this is a function known as "PARAM-EVAL" which takes an external slot name as a parameter and finds the corresponding property value of the most recent PSN process. Figure 5-1 shows how a form calling a LISP expression is represented. The form "ADD01" will return the result of adding 95 to 116. The system avoids having to have an explicit instance of the class "PROGRAM" for each primitive function by creating the forms independently of such programs. "ADD01" is therefore not a subclass of any program.
V.3 Interpreting

figure 5-1

```
ADD01
parameters
right   quote  \rightarrow 116
left    quote  \rightarrow 95
returns
r1      eval  \rightarrow (ADD (PARAM-EVAL right) (PARAM-EVAL left))
```
The basic structure of the interpreter is a loop which continues until the processes on the dynamic chain are executed to completion. At the beginning of an iteration, the process under consideration is known as the current process. Each process has associated with it a system property known as "STATE:" which consists of a list of the slots not yet executed, with prerequisite slots preceding the body slots which precede the "returns" slot. Another system property is the "CURRENT-SLOT:". If, at the beginning of an iteration, there are no slots remaining in the "STATE:" , the returned value of the current process becomes the value of the property indicated by "CURRENT-SLOT:" of the predecessor process in the dynamic chain. Also, the current process is killed, and the predecessor becomes the new current process.

If at the beginning of an iteration, the "STATE:" is not empty, the first slot in this state becomes the current slot and the state list is popped. At this point one of the following four alternatives is possible:

1. the current slot is a "quote_parameters" slot and has a quote property value; this value becomes the current slot property value of the current process;

2. the slot has an "eval" property value which is a LISP form; this is evaluated using the LISP EVAL and the returned value becomes the property value of the process;

3. the "eval" value is a PSN form; the actions are create a new process and make it an instance of the form, evaluate the parameters, assert the dynamic relation with the current process, and make the new process the current process;

4. in all other cases, the value "unknown" becomes the property value for the current slot.
There are a few special PSN objects whose associated forms perform special book-keeping actions for the interpreter. These objects will often be included in the state of a process although they do not appear in the corresponding program. Their effects involve fiddling with the relation "dynamic" and will be described in the following subsections.

V.3.3 Dynamic and Parameter Evaluation

The dynamic relation is implemented as an association list where the pairs are of the form "((<program>) . <process>)" where "<program>" is one of the types of the process. Two processes are related by "dynamic" if they are adjacent on this a-list. The implementation allows two kinds of special values in this list:

1. an atom in the a-list is called a block. When the interpreter reaches a block, instead of assigning the value to be returned to a previous process, it returns the value as a LISP function. This allows primitive functions to invoke the interpreter.

2. pairs of the form "((XXXX . <process>)" are ignored by most routines. In particular, processes on either side of such a pair are considered an assertion of "dynamic". Such pairs become regular pairs when the current slot is an instance of the class "restore". This slot causes the relation "dynamic" to be asserted between the process in the special pair and that preceding it.

The restore mechanism allows the parameter evaluation to be performed in the interpreter loop in the same way as the evaluation of other slots. The restore command is placed in the state of a process immediately after the last "parameters" slot and before any other slots. Processes created in parameter evaluation will therefore be related through the relation "dynamic" to the calling process of the process for
V. 3 Interpreting

which these values are being computed.

V. 3.4 Detaching Processes

A second special command in the state of a process is the object "&exit-slot" which causes the actions of the program "DETACH" to be performed. When this is the current slot, the following actions occur:

1. the first instance of the program which is the value of the "label" property of the current process (which is an instance of "DETACH") is disconnected from its calling process (that is, the assertion of "dynamic" between the two is removed),

2. the value of the "return" property of the current process becomes the value returned to the calling process of the detached process, and

3. the current process is killed.

Since "dynamic" is implemented as a list, the part which is removed when the link is broken in step one must be stored with the state of the process.

The final special object is known as "&break-slot" which is used to set up a generator. When this is encountered, the current process is detached, and the process object becomes the value to be returned. A break is usually placed after the prerequisite evaluations of a generator so that when the break occurs, one has a valid process. A process is made a generator only when it is created through the special class "GENERATOR" whose add program is capable of inserting the break.

V. 3.5 Exception Handling

Checking of a prerequisite is done as the attempt is made to make a value the value of the property. If the value "false" is to be assigned to a "prerequisites" slot the
exception handling mechanism becomes active. This mechanism is implemented exactly as described in chapter two.

V.4. Predefined Programs

The predefined programs are the means by which the major control structures of the PSN language are implemented. For example, the expression "new ..." results in a form calling the program "NEW".

V.4.1 NEW

Forms which call "NEW" are used to begin the process of creating an object. Once the parameters have been evaluated, the creation routine described in chapter four can be invoked. The interesting problem here is representing the parameters.

The object to be created may have an arbitrary number of types. Therefore, the form has one "instance-of" parameter which is expected to be a LISP list of forms which will evaluate to classes. Similarly, the "isa" parameter requires another list of forms which will evaluate to classes. Thus the actions of "NEW" include routines for evaluating each of the forms in these lists.

The "with" parameter of the program requires a still more complex data type. This is a list of ordered pairs of which the first element is the external name of a property, and the second element is a form which can be evaluated to produce a property value for the new object. Before "NEW" can start the creation process it must create a similar list with the values of the forms replacing the forms.

The most complex of the parameters is that which expects a description of the structure of a class. Here, a list of descriptions of slots must be passed to the creation routine. Each slot descriptions contains a number of forms each of which
V.4 Predefined Programs

must be replaced by a value by "NEW". A slot description may also contain a structure description (if the slot is a metaslot), which requires a recursive call to the structure formation routine.

V.4.2 RUN

The PSN program "RUN" is used to create a form which calls the program specified as the argument "program" with the parameters bound as specified by the argument "with". The form of the "with" argument is a list of pairs of the form

"(<parameter-name> . <form>)"

which specifies the form for the "eval" property of each parameter. Once the form is created, it is immediately interpreted. The form is then killed and the value which it produced is returned.

The need for "RUN" results from attempts to write recursive programs in PSN. When a program is created, the creation algorithm requires that all the objects (especially forms) which are to be associated with the slots of the program be created. However, a form which recursively calls a program must be a subclass of that program, and can therefore not be created before the program is created. Instead, a subclass of "RUN" is created to provide a form each time the recursive call is made. This still requires that the internal name of the program is known when the "RUN" form is created, but when these are generated by the parser, the internal name is generated by the parser.

"RUN" is also used when the program to be called is a variable.

V.4.3 Other Predefined Programs

The program "BEGIN" is supplied to implement begin-end blocks exactly as described in section 3.5. The algorithm is
simply to return the value of the parameter "arg1" if the value of the parameter "arg2" is "NULL_FORM", otherwise return the value produced by interpreting the value of "arg2".

Another important control structure implemented by a program is conditional evaluation. The program "IF" takes four parameters: "condition" which is a Boolean value, and "true", "false", and "unknown" which are forms. The value returned by "IF" is that produced by interpreting the form which is the value of the slot indicated by the value of the "condition" parameter. For example, if the value of the condition is true, the form "true" is evaluated. etc.

The final PSN control structure is the for-loop. This is implemented by forms calling two programs. The first form is a call to the program "FOR-LOOP" which has two parameters: "generator" of type "GENERATOR", and "code" which is of type "FORM". The second form is a call to the program "FOR" which has two parameters: "self" which is a form, and a slot whose value is any object. This form is the "eval" property value of the returns slot of the first form.

The set of forms which make up a typical for-loop are illustrated in by the example in figure 5-2. When the loop is entered, the "eval" property of "generator" results in the creation of a new generator. The call to "FOR-BODY" is then begun: the parameter "x" in the example illustrates the parameter which was not named above and is used as the loop variable. It gets its value through the evaluation of a call to "GET" which simply attaches the generator and interprets it. It returns the value returned when the generator detaches.

The first action in the execution of "FOR-BODY" is to apply the interpreter to the value of "code". Next, if the generator is dead (that is, no objects remain in the state of the generator) it returns the value returned by the evaluation of "code". Otherwise it returns the value returned by
interpreting the form which is the value of "self". If the value of "self" is the form of which the process is an instance, the effect will be the evaluation of the form "code" until the generator is exhausted.
CHAPTER VI
Examples

VI.1. Introduction

This chapter illustrates the use of the new language features in the writing of programs. At the present time, the implementation is not complete and therefore actual computer runs are not possible. The examples which this chapter provides were chosen because they illustrate both interesting language features and the fact that a reasonably sized implementation of the interpreter can be written within the PSN language.

VI.2. Lists

In many of the predefined programs presented in chapter five, one finds that the parameters require a list of objects. The computer implementation used the LIS? list structure for such parameters because the programs were usually implemented through LIS? functions which operate efficiently on such structures. The first example in this chapter will illustrate how lists and a routine to create them may be written in PSN. Figure 6-1 illustrates a pair of classes which implement the general list data-type. A list is either a "LIST_ELEMENT" which is an object with a pointer to an element of the list and a pointer to the next "LIST_ELEMENT", or the special object "NIL". Thus the representation for the list "(a b c)" is that illustrated in Figure 6-2. Figure 6-3 shows that one can restrict the elements of a list to be a given type by giving a less general type to the "first" slot. In this case the class "PERSON_LIST" is a class of lists whose elements are people. Figure 6-4 shows the parts of the respective programs which
VI.2 Lists

- **LIST**
  - to-test
  - returns
  - `eval`
  - `r`
  - `[instance = "NIL
  instance ? LIST_ELEMENT]`

- **LIST_ELEMENT**
  - type
  - rest
  - first

- **OBJECT**

```
figure 6-1
```

- **LIST_ELEMENT**
  - rest
  - first

```
figure 6-2
```
VI.2 Lists

Program

```
PROGRAM

LIST to-test

PERSON_LIST to-test

returns eva->[instance = NIL
instance ?
PERSON_LIST_ELEMENT]

LIST_ELEMENT

NIL

PERSON_LIST_ELEMENT

first type

OBJECT

FROM test for LIST

PARAMETERS

left eval [instance]
right quote LIST_ELEMENT

FROM test for PERSON_LIST

PARAMETERS

right quote PERSON_LIST_ELEMENT
```

Figure 6-3

Figure 6-4
change as PSM objects to demonstrate that "PERSON_LIST" is a valid specialization of "LIST".

A list set up in this way is very similar to a list in LISP. If "list" is an instance of "LIST_ELEMENT", then "list $.first" (the value of the "first" property of the object "list") corresponds to the LISP expression "(CAR list)"., and "list $.rest" corresponds to "(CDR list)". The object "NIL" is a special list element which signals the end of a list. It is defined so that "NIL $.first" and "NIL $.rest" are both equal to "NIL". The need for the special class "LIST" arises for typed lists --- one would like a unique object for "NIL", but if a list is typed, the general "NIL" will not be an instance of the list element class for that type of list (this can be seen in figure 6-3).

The following is a routine which might be used by the add program of the class "LIST". The following points should help clarify the example:

1. keywords of the language are underlined in this chapter to increase readability.

2. the keyword "program" is an abbreviation for "new PROGRAM structure" which means create a new instance of the class "PROGRAM" having the structure indicated.

3. the names "parameters", "body", and "returns" are used to indicate that the slots listed following each name are instances of that metaslot of the class "PROGRAM".

4. a slot is defined by the construction "name := a type, type, ... with etc." The construction "name:type, type, ... : form; with ..." is an abbreviation for "name := a type, type, ... with eval <- [form], ..."

5. the returns slot may not cause any side effects, so one often finds that it simply returns the value of one of the body slots (in this case "hi").
6. the construction "get generator" indicates that the generator is to run and return the next value. If the generator has been terminated, the expression fails.

```make_list := program
parameters
  q := a GENERATOR end slot
body
  b1:OBJECT:
    if (dead g)
      true
      NIL
    false
      new LIST_ELEMENT with first <- get g, rest <-
      run make_list with g <- q
    end if;
end slot
returns
  r1:OBJECT:b1; end slot
end program
```

This routine takes each element returned by the generator and includes it in the list. For a call to the add program of "LIST" (for example, "new LIST with generator <- new GENERATOR with class <- some_class end end"), the routine "make_list" would be called with the generator supplied in the "new" statement.

VI.3. Finding Variable Values

An important aspect of PSN programs is the use of slots as variables and property values as slot bindings. The example of this section is a function which, given a slot and a starting point on the dynamic chain, will find a property value corresponding to the slot, in effect locating a value bound to a variable. The function "find_slot_value" implements such a
VI.3 Finding Variable Values

find_slot_value := program
parameters
  process := a PROCESS end slot
  variable := a slot end slot
body
  b1:OBJECT:
  begin
  locals
    part := a slot end slot
  end locals
  for type_of in type process do
    if ?PART_OF :: variable ? type_of and
      part = unknown
      true
      part <- type_of
    end if
  end for
  if ?PART_OF :: variable ? part
    true
    process $ variable
  false
    eval find_slot_value with process <- for y in
dynamic(process) do y end, variable <-
  unknown
  unknown
  end if
  end begin;
end slot
returns
  r1:OBJECT:b1; end slot
end program
VI.3 Finding Variable Values

This example illustrates the following points:

1. There is no special syntax for fetching unique objects. Thus, when one desires to find the range instance of some assertion of "dynamic", it is necessary to use a for-loop (for example, "for y in dynamic(process) ...").

2. Since objects may have more than one type, operations using types must use a for-loop with a type generator ("for y in type process ..."). This operation returns only the non-redundant set of types, not the entire set.

VI.4. An Interpreter

The example of this section is a program which implements the main loop of a PSM interpreter. Several necessary aspects have been left out of the example so that the code shown may be shorter. The more important omissions are discussed in the text following the program.

```
interer := program

current_slot := a slot end slot
returning := an OBJECT end slot
return_slot := a slot end slot
previous_process := a PROCESS end slot
parameters
current_process := a PROCESS end slot
body
body_slot:OBJECT:
for y in state(current_process) do
  if y=NIL
    true
    for z in returns do
      for zz in type current_process do
        if PART_OF :: z ? zz true return_slot <- z end if
```
end for
end for;
returning <- current_process $ return_slot;
for z in dynamic(current_process) do
    previous_process <- z
end for;
kill current_process;
run assign with process <- previous_process, value <- returning;
run interpreter with current_process <- previous_process
false
current_slot <- y $ .first;
assert CURRENT_SLOT : current_process ->
current_slot;
assert state : current_process -> run
pop_list with list <- y;
if current_slot ? quote_slot
true
    run assign with process <-
current_process, value <- current_slot $ .quote, slot <- current_slot;
run interpreter with current_process <-
current_process
false
if current_slot ? action_slot
true
true
if current_slot $ .eval ? PRIMITIVE
true
run assign with process <-
current_process, value <- run
PRIM with form <-
current_slot $ .eval;
run interpreter with
current_process <-
current_process
false
run new_process with
    form <- current_slot .
    .eval, old_process <-
    current_process
unknown
run assign with value <-
        unknown, process <-
current_process
end if
false
run assign with value <-
process <-
current_process
unknown
run assign with value <-
process <-
current_process
end if
end if
end if
end for;
end slot
returns
r1:OBJECT:b1; end slot
end program

The major problem with this program is that it is a recursive program which has no termination condition, hence it will never return. A solution to this is to add a parameter which would be assigned the process which calls the interpreter, and to kill this process as the first action of interpreting. The old state of the interpreter will then not be retained.

The programs "assign" and "new_process" will not be included as examples. "assign" assigns the value given as a parameter to the current slot of the process which is the value
of the parameter "process". Its secondary function is to check that value assigned to prerequisite slots is not "false". If the prerequisite value is false, the exception handling routine described in the next section is invoked.

The routine "new_process" is used to create a new process, perform the parameter evaluations, and link the new process to the old process via "dynamic". The parameter evaluations will require that forms be interpreted, but again the fact that the interpreter never returns causes problems. This can be corrected by providing the interpreter with yet another parameter which would take as its value a process which would be a signal to halt execution. When the current process has this process as a value, a value would be returned by the interpreter to the routine which called it.

VI.5. Exception Handling

This section describes a routine which implements exception handling as described in chapter three. The program is given first:

---

exception_processor := program

the_exception := an EXCEPTION end slot
exception_type := an EXCEPTION_CLASS end slot
calling_process := a PROCESS end slot
call_handler := a FORM end slot
calling_slot := a slot end slot
the_handler := a PROGRAM end slot
parameters
bad_slot := a slot end slot
bad_process := a PROCESS end slot
body
body_slot1:OBJECT:
begin
the_handler <- run new_process with form <-
(bad_slot $ .exception), old_process <- bad_process;
the_exception <-
   interpret (bad_slot $ .exception);
for x in type the_exception do
   if x ? EXCEPTION_CLASS
      true
      exception_type <- x
   end if
end for;
for x in dynamic(bad_process) do
calling_process <- x end for;
for x in CURRENT_SLOT(calling_process) do
calling_slot <- x end for;
for x in slot do
   if PART_OF:: x ? (calling_slot $ .exception_action)
      true
      for y in type x do
         if y = exception_type 1
            exception_type subclass y
            true
            the_handler <- x $ .handler
         end if
      end for
   end if
end if
call_handler <- new FORM isa the_handler structure
   quote_parameters
      exception ::= a * with quote <-
      the_exception end slot
   end new;
kill bad_process;
run new_process with form <- call_handler,
   old_process <- calling_process
end begin
VI.5 Exception Handling

The routine "new_process" is used twice in "exception_processor" to evaluate forms. This is used instead of the "interpret" command because the new process must be attached to the user's dynamic chain, not that joining the processes of the interpreter. The second call to "new_process" is preceded by an explicit creation of a form in which one of the parameters is assigned a "quote" property. The star ("*") in the type of the slot indicates that the types are to be inherited without modification. This creation is necessary because once execution of the form calling the handler begins, any reference to the variables in "exception_processor" cannot be resolved because the variable binding mechanism will search the user's dynamic chain. The expression "isa the_handler" is what makes the form a call to the exception handler.

Once the handler has been run, control is returned to a routine (probably "assign") which must continue with the exception handling, that is decide where to return control in the user's dynamic chain. In TAPIS the prerequisite which failed would be re-evaluated and the normal sequence of control resumed.

VI.6. Using Exceptions

This section shows program fragments which associate an exception to a a prerequisite and a calling program with a corresponding exception handler link. The context used for the examples is that of an airline reservation in which a flight may have been booked to capacity (this is the context of the example in [Mylopoulos et al. 1978]).
First the exception class no seats left is associated with a corresponding prerequisite of the reservations program.

```
reserve_seat := program
...
prerequisites
  check_seats:BOOLEAN:
    flight $ .number_seats_left > 0;
    with exception <- [new NO_SEATS_LEFT with
      flight <- flight, person <- person]
  end slot
...
end program
```

The program to reserve a seat on a flight can be called by a program which arranges a trip to Europe. The next program fragment shows the association of a handler to the exception.

```
arrange_europe_trip := program
...
body
...
get_reservation:OBJECT:
  run reserve_seat with flight <- flight, person <-
    client;
  with exception_action <-
    new EXCEPTION_HANDLER_LINK structure
      eh1 := a NO_SEATS_LEFT with handler <-
        FIND_ALTERNATIVE
  end slot
  end new
end slot
...
end program
```
This piece of code attaches an exception handler link to the slot "get_reservation"; the action to be performed when "NO_SEATS_LEFT" is raised is "FIND_ALTERNATIVE" which should perform appropriate actions to maintain consistency in the knowledge base. The program "reserve_seat" will, however, be called from the form representing the call to "RUN", and from the strict interpretation of the exception handling mechanism, the handler for the exception raised in "reserve_seat" must therefore be supplied by "RUN". This problem is handled by giving the interpreter knowledge of the special forms which require that exception handler links be inherited.

This solution is especially necessary in the case of begin blocks because such structures expand into many forms. Figure 6-5 illustrates a sequence of forms in a begin-end structure for which an exception handler link is specified for the slot calling this structure. The dotted arcs illustrate the effective inheritance of the link. The exception handlers are therefore specified for each form in the block.
VI.6 Using Exceptions

Figure 6-5
CHAPTER VII

Conclusion

VII.1. Summary

This thesis consists of three parts: a brief survey of the current state of the procedural semantic network formalism, proposals for new mechanisms which are useful in organizing knowledge, and a description of a computer implementation of the formalism.

At the time of writing, the computer implementation of PSN is not yet completed. What presently exists is:

1. a parser for the language described in appendix 1 and a routine for converting the output of the parser into PSN forms;

2. standard add, fetch, test, and kill programs for relations and classes;

3. mechanisms for defining and inheriting structure and metastructure;

4. routines for asserting and retrieving slot values; and

5. an interpreter for PSN programs.

Work to complete the implementation and improve the formalism is continuing.
One of the original contributions of the PSN formalism [(Levesque 1977)] was to provide programs with declarative structure in the hope that this would simplify the programming task and make programs easier to understand. This thesis goes further in this direction by improving the way in which the declarative structure of programs interacts with the IS-A hierarchy. Programs are more easily understood because the consequences of each distinct unit (a form attached to a slot) of a program can be considered in isolation from the others. Programming for complex problems is simplified because the inheritance mechanism allows programs to be specialized to specific tasks through the specification of only the differences in the task. Finally, the intensional definition of specialization in terms of net side effects is useful in the maintenance of a consistent knowledge base.

Metastructure is introduced into PSN as an extension to the organizational tools provided by the formalism. The ability to distinguish slots and to constrain the definitions of properties found immediate use in the description of programs. The introduction of this new concept differs slightly from other extensions to PSN in that it requires a minor modification in the original PART-OF mechanism. On the other hand, one could have introduced both metastructure and new rules constraining the definition of slots defined in terms of the old mechanism which would remain in the background.

A computer implementation of the formalism provides a means by which the ideas of PSN may be tested. Once this implementation is complete it will be possible to discover the strengths and weaknesses of the formalism through the use of working knowledge bases. One will be able to test the usefulness of extensions to PSN through actual programming of these extensions. The implementation has already demonstrated the compactness of PSN because, although a large number of
primitive LIS? functions have been written, the system depends on a very small number of primitive operations which are used by the more complex tasks.

VII.3. Directions for Further Work

The problems which have yet to be solved can be divided into three areas: additions to the system and language to make it more habitable, inclusion of existing features of PSN not yet implemented, and theoretical problems and additions to the PSN formalism itself.

In the first category, an important addition would be a routine for producing a readable (paragraphed) output of the input language which can presently be printed only as a stream of tokens. A related problem is producing a readable printout of the contents of the knowledge base. One possibility in this direction is to represent objects by some input string which would create that object in the context of the knowledge base already printed. This representation would of course have to be paragraphed.

An important flaw in the language as implemented is the inability to exit for loops prematurely. This requires a structure such as "out with value <- ..." as introduced in [Levesque 1977]. For loops would also benefit from a mechanism with which more than just the index variable could change from one iteration to the next. This might be done by adding "eval" properties to the local variables which would be computed at the start of each iteration (i.e., when a new process is created). A third improvement to the language would be a mechanism for attaching exception handler links to the individual forms in a begin block.

Many aspects of the PSN formalism have not yet been successfully implemented. The exception handling mechanism and IS-A inheritance between forms have not yet been tested. The
behaviour of the third Boolean value, "unknown", has not yet been specified.

The restriction mechanism of PSN has not yet been incorporated into the implementation. Questions that have yet to be answered for restrictions are:

1. What is a restriction? Is a restriction a predicate, and if so, how many arguments does it have? Alternatively, a restriction might be a form. In this case it must be decided what slots it may reference.

2. When is a restriction applied? The restrictions on the slots of a class could all be applied to an instance before the add program is run or when the add program is complete. Alternatively, a slot restriction could be applied when a value is assigned to the slot.

3. What does the restriction mean? The restriction could cause instantiation to fail if an incorrect value is given for a property. Restrictions could also be used to compute the values of a slot given values for the other slots.

Newer aspects of PSN which have yet to be included in the implementation are the lattice restrictions and automatic location of meet classes described [Schneider 78a]; the handling of indefinite objects described by D. Berlin ([Berlin 1979]); and the static exception handling mechanisms described in [Lesperance 1980]. Also the concept of context which has been an important theme in PSN ([Levesque 1977], [Schneider 1978a], [Schneider 1979]) has not yet been included in the implementation.

Finally there are problems within the formalism which require attention. One would like the inheritance of property values to be stronger and apply a uniform IS-A constraint on
such values. At present there are examples in which the inheritance of add programs fail ([Levesque and Mylopoulos 1979]). Another interesting direction of research would provide mechanisms with which higher levels of metaobjects could constrain distant descendants in the INSTANCE-OF hierarchy. At present the constraints provided by the metastructure of a metaclass control only the structure of instances and the property values of instances of the instances.

VII.4. Concluding Remarks

Although a number of problems remain to be solved, both within the formalism and in the implementation, PSNI is evolving into a powerful tool for the representation of knowledge.
Bibliography

[Berlin 1978]
Berlin, D. A description of a computer implementation of a PSN system. Internal Memo, Department of Computer Science, University of Toronto, 1978.

[Berlin 1979]

[Graham and Kramer 1979]

[Gries 1971]

[Kidd et al. 1977]
Kidd, R., Schneider, P., Vassiliou, Y. "Using AI to Fill Out Your Individual Income Tax Return." AI Memo 77-2, Department of Computer Science, University of Toronto, April 1977.

[Kramer 1979]
Kramer, B. M. "Representation of Programs" in "Topics in a Procedural Semantic Network Formalism." AI-MEMO 79-3, Department of Computer Science, University of Toronto, June 1979.
[Lesperance 1979]
Lesperance, Y. Description of an experiment in modelling a consumer’s world using an implementation of PSN. Internal Memo, Department of Computer Science, University of Toronto, 1979.

[Lesperance 1980]

[Levesque 1977]

[Levesque and Mylopoulos 1979]

[Mylopoulos et al. 1978]

[Schneider 1978a]

[Schneider 1978b]

[Schneider 1979]

[Wong 1980]
A1.1. Introduction

This appendix gives a complete syntax for the PSN language currently supplied by the implementation. Each section includes some notes on the meanings of the various constructs.

The notation used for describing the language is a modified Bachus-Naur form for which the differences are:

1. The metasymbols are ' = {}[ ]''**, and the blank.
2. Terminals are always enclosed in quotes ("), whereas a non-terminal is any string of characters delimited by any of the metasymbols.
3. The metasymbol ':' is used to replace the '::=' symbol.
4. Concatenation is indicated by blanks and has a higher precedence than alternation which is indicated by commas. Except at the top level of a rule, a set of alternatives is enclosed in braces ('[]').
5. Zero or one occurrence of a term is enclosed in square brackets ('[ ]').

A1.2. Names and Constants

In the input stream a name is a sequence of characters beginning with a letter and followed by any number of letters, digits, or underscores (''). Any other character will stop the scanning of a name. A name in a PSN program has one of two
meanings: it is either a reference to a slot value if it names a slot defined in some program statically containing the form in which the name appears, or it is an external name of some object in the knowledge base.

Constants which are allowed in PSN are the tokens "true", "false", and "unknown", strings, and numbers. A string is a sequence of any characters except the quote ("), which is enclosed in quotes. A number is a sequence of digits which may optionally include one period representing the decimal point.

A1.3. Expressions

This section describes the various forms of expressions. PSN supplies the normal set of arithmetic and logical expressions. In addition, there are several operations unique to the language. These are described briefly as follows:

1. "x$slot" - returns the property value of object "x" for "slot" which is either a slot name or a slot object.
2. "get generator" - returns next value computed by the generator. If the generator is exhausted (dead), the operation fails.
3. "eval program with p1<x ..." - run the program with parameter values as supplied.
4. "interpret form" - like "eval" but a form already has its parameters bound.
5. ".name" - this quoting operation returns the slot object rather than the value indicated by the name.
6. "[form]" - returns the form object without interpreting.
7. "*" means inherit something from the corresponding position in an IS-A parent (useable only if the inheritance operation is well defined).
Expressions may also be Boolean expressions. The operators for and, or, and not must deal with a three-valued logic. In addition to the regular predicates which compare the values of numbers, there are several special PSN predicates:

1. "dead x" - the value is true if x is a process and its state is empty.
2. "x?y" - applies the test program of "y" to object "x".
3. "r::x?y" - applies the test program of relation "r" to the pair of objects "x" and "y".
4. "x subclass y" - tests if "x" is a subclass of "y".

expression = e1 ("|" e1)*.

e1 = e2 ("&" e2)*.

e2 = e3 ("=" e3, 
    
  " =", 
  
  "<", 
  "<=",
  
  ">",
  
  ">="} e3)*.

e3 = e4 ("+", 
    
  "-" e4)*.

e4 = e5 ("*", 
    
  "/" e5)*.
A1.3 Expressions

\[ o_5 = \text{"eval" } e_6 \ [\text{"with" } \text{with-clause}], \]
\[ \text{"get" } e_6, \]
\[ \text{"+" } e_6, \]
\[ \text{"-" } e_6, \]
\[ \text{"" } e_6, \]
\[ \text{"interpret" } e_6, \]
\[ \text{"dead" } e_6, \]
\[ e_6. \]

\[ e_6 = e_7 \ \text{"$" } e_7, \]
\[ e_7 \ \text{"?" } e_7, \]
\[ e_7 \ \text{"::" } e_7 \ \text{"?" } e_7, \]
\[ e_7 \ \text{"subclass" } e_7, \]
\[ e_7. \]

\[ e_7 = \text{"\$"}, \]
\[ \text{"{" } \text{expression "")"}, \]
\[ \text{".\" } \text{name}, \]
\[ \text{string}, \]
\[ \text{number}, \]
\[ \text{name}, \]
\[ \text{"true"}, \]
\[ \text{"false"}, \]
\[ \text{"unknown"}, \]
\[ \text{"[" } \text{expression ""]"}, \]
\[ \text{interval}. \]

A1.4. Statements

Statements differ from expressions only in that they may have net side effects. All statements return values and an expression may appear anywhere that a statement may appear.

The operator \":=\" makes the left side an external name of the object returned by the right side. If the left side begins with "a" or "an", a slot is created with the external name
indicated. This latter operation may occur only within a structure.

The "begin-sub" construct allows special statements to indicate that inheritance is to take place from an IS-A parent. The statement "*" means inherit one statement in the corresponding position of the parent. ".." means inherit from this point until the end of the original begin block. Statements following this construct become a sequence of forms replacing the "NULL_FORM" which terminated the original block.

\[
\text{primary-action} = \text{name} \text{"\:=\"} \text{action}, \\
\quad \text{action} \text{"\<\−\"} \text{action}, \\
\quad \text{action}.
\]

\[
\text{action} = \text{create}, \\
\quad \text{begin}, \\
\quad \text{run}, \\
\quad \text{assert}, \\
\quad \text{if}, \\
\quad \text{for}, \\
\quad \text{remove}, \\
\quad \text{kill}, \\
\quad \text{instanceof}, \\
\quad \text{unassert}, \\
\quad \text{"print" expression}, \\
\quad \text{expression}.
\]

\[
\text{assert} = \text{"assert" expression \"\:=\" expression \"\−\→\" expression}.
\]

\[
\text{begin} = \text{"begin" ["locals" slot-definition+] begin-sub}.
\]

\[
\text{begin-sub} = \text{"end"}, \\
\quad \text{primary-action "end"}, \\
\quad \text{primary-action \"\:=\" begin-sub}, \\
\quad \"..\" begin-sub, \\
\quad \";\" begin-sub,
\]
A1.4 Statements

"run" begin-sub.

run = "run" expression ["with" with-clause].

if = "if" expression ["true" begin-sub] ["false" begin-sub]
    ["unknown" begin-sub] "end" ["if"].

for = "for" name generator "do" ["locals" slot-definition+]
  begin-sub.

generator = "in" expression ["(" expression ")", "(" ",
    expression ")")"],
  "in" "type" expression.

unassert = "unassert" expression ":" expression "->" expression.

remove = "remove" expression "from" expression.

kill = "kill" expression.

instanceof = "make" expression "instanceof" expression ["with" with-clause].

A1.5. Creation

These rules describe the statements one would use to create objects.

create = "new" expression "," expression)* create-mods "end"
    ["new"],
  "newsub" expression "," expression)* create-mods
    "end" ["new"],
  "program" create-mods "end" ["program"].
create-mods = "with" with-clause create-mods,
   "structure" structure-definition create-mods,
   "isa" expression ("","" expression)* create-mods,
   lambda.

with-clause = name "<-" expression ("","" with-clause)*.

interval = "<" number "," {number, "*"} ">".

structure-definition = [ name ] slot-definition+
   structure-definition,
   lambda.

slot-definition = name ":=" {"a", "an"} expression ("","" expression)* slot-mods,
   name ":=" "newmetaslot" slot-mods,
   name ":=" expression ("","" expression)* ":"
   primary-action ";" slot-mods.

slot-mods = "with" with-clause slot-mods,
   "structure" structure-definition slot-mods,
   "isa" name (""," name)* slot-mods,
   "end" [{"a", "slot", "metaslot", "newmetaslot"}].
Index

Arc ... ... 10
Assertion ... ... 25
Assertional properties ... ... 25
Begin - end ... ... 52
Block ... ... 86
Body ... ... 39
BEGIN ... ... 50
Class ... ... 2, 11
Complex properties ... ... 8
Context ... ... 11
CLASS ... ... 11
CURRENT-SLOT: ... ... 85
Declarative ... ... 10
Default ... ... 16
Detach ... ... 50
Domain ... ... 26
Domain_interval ... ... 28
Dynamic ... ... 48
Eval property ... ... 41
Exception ... ... 54
Exception handler ... ... 54
Exception property ... ... 55
Exception-action property ... ... 55
Excuse ... ... 54
Expression ... ... 44
External ... ... 2
External name ... ... 11
EXCEPTION-HANDLER-LINK ... ... 55
EXCEPTION_CLASSES ... ... 55
Fail ... ... 42
Form ... ... 41, 44
FORM ... ... 41, 82
Index

Generator ... 37, 50
Inheritance ... 19
Instance ... 2, 11, 25
Intensional ... 64
Internal ... 63
Interval ... 28, 64
INSTANCE-OF ... 10
IS-A ... 4, 11, 19
IS-A hierarchy ... 22
Legal specialization ... 39
Link ... 10
List ... 93
Meta class ... 14
Metameta class ... 34
Metaslot ... 33
Metastructure ... 3, 30, 34
METACLASS ... 31
Met side effect ... 36
Mode ... 10
Non-redundant type set ... 65
NULL_FORM ... 50
NULL_PROGRAM ... 50
Object ... 2
OBJECT ... 15, 69
Prerequisites ... 39
Primitive ... 61
Procedural attachment ... 17
Process ... 42
Program ... 36
Property categories ... 8
PARAM-EVAL ... 83
PART-OF ... 10, 16
PROGRAM ... 41
Quote property ... 48
Quote_parameters ... 48
Raising ... 54
Range ... 26
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range_interval</td>
<td>28</td>
</tr>
<tr>
<td>Redefinition</td>
<td>25</td>
</tr>
<tr>
<td>Relations</td>
<td>25</td>
</tr>
<tr>
<td>Restriction</td>
<td>15</td>
</tr>
<tr>
<td>Returns</td>
<td>39</td>
</tr>
<tr>
<td>RELATION</td>
<td>26</td>
</tr>
<tr>
<td>Scanner</td>
<td>79</td>
</tr>
<tr>
<td>Side effect</td>
<td>36</td>
</tr>
<tr>
<td>Slot</td>
<td>3, 15</td>
</tr>
<tr>
<td>Slot value</td>
<td>15</td>
</tr>
<tr>
<td>Slot-spec</td>
<td>64</td>
</tr>
<tr>
<td>Source</td>
<td>26</td>
</tr>
<tr>
<td>Specialization</td>
<td>19</td>
</tr>
<tr>
<td>Structural property definition</td>
<td>15</td>
</tr>
<tr>
<td>Structural property value</td>
<td>15</td>
</tr>
<tr>
<td>Structure</td>
<td>3, 15</td>
</tr>
<tr>
<td>Subclass</td>
<td>4, 19</td>
</tr>
<tr>
<td>Superclass</td>
<td>19</td>
</tr>
<tr>
<td>STATE:</td>
<td>85</td>
</tr>
<tr>
<td>Target</td>
<td>26</td>
</tr>
<tr>
<td>To-add</td>
<td>2, 13</td>
</tr>
<tr>
<td>To-assert</td>
<td>26</td>
</tr>
<tr>
<td>To-fetch</td>
<td>14</td>
</tr>
<tr>
<td>To-kill</td>
<td>13</td>
</tr>
<tr>
<td>To-test</td>
<td>13</td>
</tr>
<tr>
<td>Transaction</td>
<td>44</td>
</tr>
<tr>
<td>Type</td>
<td>3, 11, 15</td>
</tr>
<tr>
<td>Type set</td>
<td>64</td>
</tr>
<tr>
<td>TAXIS</td>
<td>7</td>
</tr>
<tr>
<td>Unique redefinition</td>
<td>25</td>
</tr>
<tr>
<td>Value inheritance</td>
<td>20</td>
</tr>
</tbody>
</table>
AI Memos are occasional working papers produced by members of the Artificial Intelligence group at the University of Toronto. Here is a list of those produced to date:


16. Yves Lesperance, Bryan M. Kramer, and Peter F. Schneider. "Topics in PSN - II: Handling Exceptional Conditions in PSN; Representing Programs in PSN; Contexts in PSN". AI Memo 80-2, April 1980.
Here is a list of the theses in Artificial Intelligence produced at the University of Toronto since 1974:


BIBLIOGRAPHY OF CSRG TECHNICAL REPORTS+

* CSRG-1 EMPIRICAL COMPARISON OF LR(k) AND PRECEDENCE PARSERS
  J.J. Horning and W.R. Lalonde, September 1970
  [ACM SIGPLAN Notices, November 1970]

* CSRG-2 AN EFFICIENT LALR PARSER GENERATOR
  W.R. Lalonde, February 1971

* CSRG-3 A PROCESSOR GENERATOR SYSTEM
  J.D. Gorrie, February 1971

* CSRG-4 DYLAN USER'S MANUAL
  P.E. Bonzon, March 1971

CSRG-5 DIAL - A PROGRAMMING SYSTEM FOR INTERACTIVE ALGEBRAIC MANIPULATION
  Alan C.M. Brown and J.J. Horning, March 1971

CSRG-6 ON DEADLOCK IN COMPUTER SYSTEMS
  Richard C. Holt, April 1971
  [Ph.D. Thesis, Dept. of Computer Science, Cornell University, 1971]

CSRG-7 THE STAR-RING SYSTEM OF LOOSELY COUPLED DIGITAL DEVICES
  John Neill Thomas Potvin, August 1971

* CSRG-8 FILE ORGANIZATION AND STRUCTURE
  G.M. Stacey, August 1971

CSRG-9 DESIGN STUDY FOR A TWO-DIMENSIONAL COMPUTER-ASSISTED
  ANIMATION SYSTEM
  Kenneth B. Evans, January 1972

* CSRG-10 HOW A PROGRAMMING LANGUAGE IS USED
  William Gregg Alexander, February 1972

* CSRG-11 PROJECT SUE STATUS REPORT
  J.W. Atwood (ed.), April 1972

+ Abbreviations:
  DCS - Department of Computer Science, University of Toronto
  EE - Department of Electrical Engineering, University of Toronto
* - Out of print
* CSRG-12 THREE DIMENSIONAL DATA DISPLAY WITH HIDDEN LINE REMOVAL  
Rupert Bramall, April 1972  

* CSRG-13 A SYNTAX DIRECTED ERROR RECOVERY METHOD  
Lewis R. James, May 1972  

CSRG-14 THE USE OF SERVICE TIME DISTRIBUTIONS IN SCHEDULING  
Kenneth C. Sevcik, May 1972  

CSRG-15 PROCESS STRUCTURING  
J.J. Horning and B. Randell, June 1972  
[ACM Computing Surveys, March 1972]

CSRG-16 OPTIMAL PROCESSOR SCHEDULING WHEN SERVICE TIMES ARE HYPEREXPONENTIALLY DISTRIBUTED AND PREEMPTION OVERHEAD IS NOT NEGLIGIBLE  
Kenneth C. Sevcik, June 1972  
[Proceedings of the Symposium on Computer-Communication, Networks and Teletraffic, Polytechnic Institute of Brooklyn, 1972]

* CSRG-17 PROGRAMMING LANGUAGE TRANSLATION TECHNIQUES  
W.M. McKeeman, July 1972

CSRG-18 A COMPARATIVE ANALYSIS OF SEVERAL DISK SCHEDULING ALGORITHMS  
C.J.M. Turnbull, September 1972

CSRG-19 PROJECT SUE AS A LEARNING EXPERIENCE  
K.C. Sevcik et al, September 1972  
[Proceedings AFIPS Fall Joint Computer Conference, v. 41, December 1972]

* CSRG-20 A STUDY OF LANGUAGE DIRECTED COMPUTER DESIGN  
David B. Wortman, December 1972  
[Ph.D. Thesis, Computer Science Department, Stanford University, 1972]

CSRG-21 AN APL TERMINAL APPROACH TO COMPUTER MAPPING  
R. Kvaternik, December 1972  

* CSRG-22 AN IMPLEMENTATION LANGUAGE FOR MINICOMPUTERS  
G.G. Kalmar, January 1973  

CSRG-23 COMPILER STRUCTURE  
W.M. McKeeman, January 1973  
[Proceedings of the USA-Japan Computer Conference, 1972]
* CSRG-24 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
  J.D. Gannon (ed.), March 1973

CSRG-25 THE INVESTIGATION OF SERVICE TIME DISTRIBUTIONS
  Eleanor A. Lester, April 1973

* CSRG-26 PSYCHOLOGICAL COMPLEXITY OF COMPUTER PROGRAMS:
  AN INITIAL EXPERIMENT
  Larry Weissman, August 1973

* CSRG-27 STRUCTURED SUBSETS OF THE PL/I LANGUAGE
  Richard C. Holt and David B. Wortman, October 1973

* CSRG-28 ON REDUCED MATRIX REPRESENTATION OF LR(k) PARSER TABLES
  Marc Louis Joliat, October 1973

* CSRG-29 A STUDENT PROJECT FOR AN OPERATING SYSTEMS COURSE
  B. Czarnik and D. Tsichritzis (eds.), November 1973

* CSRG-30 A PSEUDO-MACHINE FOR CODE GENERATION
  Henry John Pasko, December 1973

* CSRG-31 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING

* CSRG-32 SCHEDULING MULTIPLE RESOURCE COMPUTER SYSTEMS
  E.D. Lazowska, May 1974

* CSRC-33 AN EDUCATIONAL DATA BASE MANAGEMENT SYSTEM
  F. Lochovsky and D. Tsichritzis, May 1974
  [INFOR, 14 (3), pp.270-278, 1976]

* CSRG-34 ALLOCATING STORAGE IN HIERARCHICAL DATA BASES
  P. Bernstein and D. Tsichritzis, May 1974
  [Information Systems Journal, v.1, pp.133-140]

* CSRG-35 ON IMPLEMENTATION OF RELATIONS
  D. Tsichritzis, May 1974

* CSRG-36 SIX PL/I COMPILERS
  D.B. Wortman, P.J. Khaiat, and D.M. Lasker, August 1974
  [Software Practice and Experience, v.6, n.3, July-Sept. 1976]

* CSRG-37 A METHODOLOGY FOR STUDYING THE PSYCHOLOGICAL COMPLEXITY
  OF COMPUTER PROGRAMS
  Laurence M. Weissman, August 1974
* CSRG-38 AN INVESTIGATION OF A NEW METHOD OF CONSTRUCTING SOFTWARE
  David M. Lasker, September 1974

CSRG-39 AN ALGEBRAIC MODEL FOR STRING PATTERNS
  Glenn F. Stewart, September 1974

* CSRG-40 EDUCATIONAL DATA BASE SYSTEM USER'S MANUAL
  J. Klebanoff, F. Lochovsky, A. Rozitis, and
  D. Tsichritzis, September 1974

* CSRG-41 NOTES FROM A WORKSHOP ON THE ATTAINMENT OF
  RELIABLE SOFTWARE
  David B. Wortman (ed.), September 1974

* CSRG-42 THE PROJECT SUE SYSTEM LANGUAGE REFERENCE MANUAL
  B.L. Clark and F.J.B. Ham, September 1974

* CSRG-43 A DATA BASE PROCESSOR
  E.A. Ozkarahan, S.A. Schuster and K.C. Smith,

* CSRG-44 MATCHING PROGRAM AND DATA REPRESENTATION TO A
  COMPUTING ENVIRONMENT
  Eric C.R. Hehner, November 1974

* CSRG-45 THREE APPROACHES TO RELIABLE SOFTWARE; LANGUAGE DESIGN,
  DYADIC SPECIFICATIONS, COMPLEMENTARY SEMANTICS
  J.E. Donahue, J.D. Gannon, J.V. Guttag and
  J.J. Horning, December 1974

CSRG-46 THE SYNTHESIS OF OPTIMAL DECISION TREES FROM
  DECISION TABLES
  Helmut Schumacher, December 1974

* CSRG-47 LANGUAGE DESIGN TO ENHANCE PROGRAMMING RELIABILITY
  John D. Gannon, January 1975

* CSRG-48 DETERMINISTIC LEFT TO RIGHT PARSING
  Christopher J.M. Turnbull, January 1975

* CSRG-49 A NETWORK FRAMEWORK FOR RELATIONAL IMPLEMENTATION
  D. Tsichritzis, February 1975 [in Data Base Description,
  Dongue and Nijssen (eds.), North Holland Publishing Co.]
CSRG-50 A UNIFIED APPROACH TO FUNCTIONAL DEPENDENCIES AND RELATIONS
P.A. Bernstein, J.R. Swenson and D.C. Tsichritzis
February 1975 [Proceedings of the ACM SIGMOD Conference, 1975]

CSRG-51 ZETA: A PROTOTYPE RELATIONAL DATA BASE MANAGEMENT SYSTEM
M. Brodie (ed), February 1975 [Proceedings Pacific ACM Conference, 1975]

CSRG-52 AUTOMATIC GENERATION OF SYNTAX-REPAIRING AND PARAGRAPHING PARSERS
David T. Barnard, March 1975

CSRG-53 QUERY EXECUTION AND INDEX SELECTION FOR RELATIONAL DATA BASES
J.H. Gilles Farley and Stewart A. Schuster, March 1975

CSRG-54 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING

CSRG-55 STRUCTURED SUBSETS OF THE PL/1 LANGUAGE
Richard C. Holt and David B. Wortman, May 1975

CSRG-56 FEATURES OF A CONCEPTUAL SCHEMA
D. Tsichritzis, June 1975 [Proceedings Very Large Data Base Conference, 1975]

CSRG-57 MERLIN: TOWARDS AN IDEAL PROGRAMMING LANGUAGE
Eric C.R. Hehner, July 1975
see Acta Informatica Col.10, No.3, pp.229-243, 1978

CSRG-58 ON THE SEMANTICS OF THE RELATIONAL DATA MODEL

CSRG-59 THE SPECIFICATION AND APPLICATION TO PROGRAMMING OF ABSTRACT DATA TYPES
John V. Guttag, September 1975

CSRG-60 NORMALIZATION AND FUNCTIONAL DEPENDENCIES IN THE RELATIONAL DATA BASE MODEL
Phillip Alan Bernstein, October 1975

CSRG-61 LSL: A LINK AND SELECTION LANGUAGE
D. Tsichritzis, November 1975 [Proceedings ACM SIGMOD Conference, 1976]
• CSRG-62 COMPLEMENTARY DEFINITIONS OF PROGRAMMING LANGUAGE SEMANTICS
James E. Donahue, November 1975

CSRG-63 AN EXPERIMENTAL EVALUATION OF CHESS PLAYING HEURISTICS
Lazlo Sugar, December 1975

CSRG-64 A VIRTUAL MEMORY SYSTEM FOR A RELATIONAL ASSOCIATIVE PROCESSOR
S.A. Schuster, E.A. Ozkarahan, and K.C. Smith,

CSRG-65 PERFORMANCE EVALUATION OF A RELATIONAL ASSOCIATIVE PROCESSOR
E.A. Ozkarahan, S.A. Schuster, and K.C. Sevcik,

CSRG-66 EDITING COMPUTER ANIMATED FILM
Michael D. Tilson, February 1976

CSRG-67 A DIAGRAMMATIC APPROACH TO PROGRAMMING LANGUAGE SEMANTICS
James R. Cordy, March 1978

• CSRG-68 A SYNTHETIC ENGLISH QUERY LANGUAGE FOR A RELATIONAL ASSOCIATIVE PROCESSOR
L. Kerschberg, E.A. Ozkarahan, and J.E.S. Pacheco,
April 1976

CSRG-69 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING

• CSRG-70 A TAXONOMY OF DATA MODELS
L. Kerschberg, A. Klug, and D.Tsichritzis, May 1976
[Proceedings Very Large Data Base Conference, 1976]

• CSRG-71 OPTIMIZATION FEATURES FOR THE ARCHITECTURE OF A DATA BASE MACHINE

• CSRG-72 THE RELATIONAL DATA BASE SYSTEM OMEGA - PROGRESS REPORT
H.A. Schmid (ed.), P.A. Bernstein (ed.), B. Arlow,
R. Baker and S. Pozgaj, July 1976
CSRG-73 **AN ALGORITHMIC APPROACH TO NORMALIZATION OF RELATIONAL DATA BASE SCHEMAS**

P.A. Bernstein and C. Beeri, September 1976

**CSRG-74 A HIGH-LEVEL MACHINE-ORIENTED ASSEMBLER LANGUAGE FOR A DATA BASE MACHINE**

E.A. Ozkarahan and S.A. Schuster, October 1976

**CSRG-75 DO CONSIDERED OD: A CONTRIBUTION TO THE PROGRAMMING CALCULUS**

Eric C.R. Hehner, November 1976

*Acta Informatica* to appear 1979

**CSRG-76 SOFTWARE HUT: A COMPUTER PROGRAM ENGINEERING PROJECT IN THE FORM OF A GAME**

J.J. Horning and D.B. Wortman, November 1976

[[IEEE Transactions on Software Engineering, v.SE-3, n.4, July 1977]]

**CSRG-77 A SHORT STUDY OF PROGRAM AND MEMORY POLICY BEHAVIOUR**

G. Scott Graham, January 1977

**CSRG-78 A PANACHE OF DBMS IDEAS**

D. Tsichritzis (ed.), February 1977

**CSRG-79 THE DESIGN AND IMPLEMENTATION OF AN ADVANCED LALR PARSE TABLE CONSTRUCTOR**

David H. Thompson, April 1977


**CSRG-80 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING**


**CSRG-81 PROGRAMMING METHODOLOGY: AN ANNOTATED BIBLIOGRAPHY FOR IFIP WORKING GROUP 2.3**


**CSRG-82 NOTES ON EUCLID**

edited by W. David Elliot and David T. Barnard, August 1977

**CSRG-83 TOPICS IN QUEUEING NETWORK MODELING**

edited by G. Scott Graham, July 1977

**CSRG-84 TOWARD PROGRAM ILLUSTRATION**

Edward Yarwood, September 1977


**CSRG-85 CHARACTERIZING SERVICE TIME AND RESPONSE TIME DISTRIBUTIONS IN QUEUEING NETWORK MODELS OF COMPUTER SYSTEMS**

Edward D. Lazowska, September 1977

CSRG-86 MEASUREMENTS OF COMPUTER SYSTEMS FOR QUEUEING NETWORK MODELS
Martin G. Kienzle, October 1977

CSRG-87 'OLGA' LANGUAGE REFERENCE MANUAL

CSRG-88 USING A GRAMMATICAL FORMALISM AS A PROGRAMMING LANGUAGE
Brad A. Silverberg, January 1978

CSRG-89 ON THE IMPLEMENTATION OF RELATIONS: A KEY TO EFFICIENCY
Joachim W. Schmidt, January 1978

CSRG-90 DATA BASE MANAGEMENT SYSTEM USER PERFORMANCE
Frederick H. Lochovsky, April 1978

CSRG-91 SPECIFICATION AND VERIFICATION OF DATA BASE SEMANTIC INTEGRITY
Michael Lawrence Brodie, April 1978

CSRG-92 STRUCTURED SOUND SYNTHESIS PROJECT (SSSP): AN INTRODUCTION

CSRG-93 A DEVICE-INDEPENDENT, GENERAL-PURPOSE GRAPHICS SYSTEM IN A MINICOMPUTER TIME-SHARING ENVIRONMENT
William T. Reeves, August 1979

CSRG-94 ON THE AXIOMATIC VERIFICATION OF CONCURRENT ALGORITHMS
Christian Lengauer, August 1978

CSRG-95 PISA: A PROGRAMMING SYSTEM FOR INTERACTIVE PRODUCTION OF APPLICATION SOFTWARE
Rudolf Marty, August 1978

CSRG-96 ADAPTIVE MICROPROGRAMMING AND PROCESSOR MODELING
Walter G. Rosocha
[Ph.D. Thesis, EE, August 1978]

CSRG-97 DESIGN ISSUES IN THE FOUNDATION OF A COMPUTER-BASED TOOL FOR MUSIC COMPOSITION
William Buxton
CSRG-98  THEORY OF DATABASE MAPPINGS  
  Anthony C. Klug  

CSRG-99  HIERARCHICAL COROUTINES: A MECHANISM FOR IMPROVED PROGRAM STRUCTURE  
  Leonard I. Vanek, February 1979

CSRG-100  TOPICS IN PERFORMANCE EVALUATION  
  G. Scott Graham (ed.), July 1979

CSRG-101  A PANACHE OF DBMS IDEAS II  
  F.H. Lochovsky (ed.), May 1979

CSRG-102  A SIMPLE SET THEORY FOR COMPUTING SCIENCE  
  Eric C.R. Hehner, May 1979

CSRG-103  THE CENTRALIZED ALGORITHM IN DISTRIBUTED SYSTEMS  
  Ernest J.H. Chang  

CSRG-104  ELIMINATING THE VARIABLE FROM DIJKSTRA'S MINI-LANGUAGE  
  D. Hugh Redelmeier, July 1979

CSRG-105  A LANGUAGE FACILITY FOR DESIGNING INTERACTIVE DATABASE-INTENSIVE APPLICATIONS  
  John Mylopoulos, Philip A. Bernstein, Harry K.T. Wong, July 1979

CSRG-106  ON APPROXIMATE SOLUTION TECHNIQUES FOR QUEUEING NETWORK MODELS OF COMPUTER SYSTEMS  
  Satish Kumar Tripathi, July 1979

CSRG-107  A FRAMEWORK FOR VISUAL MOTION UNDERSTANDING  
  John K. Tsotsos, John Mylopoulos, H. Dominic Covvey Steven W. Zucker, DCS, June 1979

CSRG-108  DIALOGUE ORGANIZATION AND STRUCTURE FOR INTERACTIVE INFORMATION SYSTEMS  
  John Leonard Barron  

CSRG-109  A UNIFYING MODEL OF PHYSICAL DATABASES  
  D.S. Batory, C.C. Gotlieb, April 1980

CSRG-110  OPTIMAL FILE DESIGNS AND REORGANIZATION POINTS  
  D.S. Batory, April 1980

CSRG-111  A PANACHE OF DBMS IDEAS III  
  D. Tsichritzis (ed.), April 1980
CSRG-112 TOPICS IN PSN - II: EXCEPTIONAL CONDITION HANDLING IN PSN; REPRESENTING PROGRAMS IN PSN; CONTENTS IN PSN
Yves Lesperance, Byran M. Kramer, Peter F. Schneider April, 1980

CSRG-113 SYSTEM-ORIENTED MACRO-SCHEDULING
C.C. Gottlieb and A. Schonbach
May 1980

CSRG-114 A FRAMEWORK FOR VISUAL MOTION UNDERSTANDING
John Konstantine Tsotsos

CSRG-115 SPECIFICATION OF CONCURRENT EUCLID
James R. Cordy and Richard C. Holt
July 1980

CSRG-116 THE REPRESENTATION OF PROGRAMS IN THE PROCEDURAL SEMANTIC NETWORK FORMALISM
Byran M. Kramer