July 1984

CADRS: A System for the Design of Complex Data Structures in LISP

Richard J. Duncan
University of Pennsylvania

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation
http://repository.upenn.edu/cis_reports/660


This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/660
For more information, please contact repository@pobox.upenn.edu.
Abstract
This thesis describes a project to design and implement a data abstraction facility for LISP. The result of this project, the CADRS (Compiler for Access of Data Regardless of Structure) system, supports the definition, creation, and efficient manipulation of hybrid data/procedure structures that closely resemble class objects in languages like Simula [2] and Smalltalk.

Comments
CADRS: A SYSTEM FOR
THE DESIGN OF COMPLEX DATA
STRUCTURES IN LISP

Richard J. Duncan
MS-CIS-83-19

Department of Computer and Information Science
Moore School/D2
University of Pennsylvania
Philadelphia, PA 19104

July 1984
A thesis presented to the Faculty of Engineering and Applied Science of the University of Pennsylvania in partial fulfillment of the requirements for the degree of Master of Science in Engineering for graduate work in Computer and Information Science.

Timothy W. Finin, Thesis Supervisor

O. Peter Buneman, Graduate Group Chairperson
Acknowledgements

I would like to thank a number of people for their help in this effort. First and foremost to my advisor Dr Timothy W. Finin for his invaluable help in the formulation of the concepts that went into the design of CADRS and for his enduring enthusiasm in the work; to the faculty and students in the graduate department for always keeping the door open when an answer was needed, especially Marty Borowski for time spent proof-reading; and finally to my family and friends for their support.
## CONTENTS

1. Introduction ................................................................. 1
   1.1 Introduction .................................................................. 1
   1.2 An Overview of this thesis ............................................. 1
   1.3 An Overview of Data Abstraction Schemes ....................... 2
      1.3.1 Basic Data Abstraction ........................................... 2
      1.3.2 Object Oriented Data Abstraction ............................ 6
   1.4 A Short Overview of CADRS ........................................... 8
   1.5 Comparative Analysis ................................................. 12
      1.5.1 DEFSTRUCT ....................................................... 13
      1.5.2 INTERLISP Record Package ................................... 16
      1.5.3 GLISP ............................................................ 18
   1.6 CADRS and Representation Schemes ............................... 19

2. CADRS Reference ............................................................. 22
   2.1 Introduction ............................................................... 22

3. structure - Structure Definition ......................................... 23
   3.1 Syntax ......................................................................... 23
   3.2 Structure Attributes .................................................... 24
      3.2.1 type .................................................................. 24
      3.2.2 fields .................................................................. 27
      3.2.2.1 examples ........................................................ 27
      3.2.3 create related attributes ....................................... 28
      3.2.3.1 to-create ......................................................... 29
      3.2.3.2 before-creating ............................................... 29
      3.2.3.3 after-creating ................................................ 29
      3.2.4 Structure typing ................................................... 30
      3.2.4.1 type? ............................................................. 31
      3.2.4.2 typecheck ....................................................... 31
      3.2.5 is-a .................................................................. 32

4. Field Definition ............................................................... 33
   4.1 Field-list specification .................................................. 33
   4.2 Syntax ......................................................................... 34
   4.3 field Attributes .......................................................... 35
      4.3.1 is-a .................................................................. 35
      4.3.2 default .............................................................. 36
      4.3.3 fetch and replace definition .................................... 36
      4.3.3.1 to-fetch ......................................................... 37
3 July 1984

12. Conclusions ........................................................................ 83
   12.1 Concluding Remarks .................................................. 83
   12.2 Future Work ............................................................... 83
1 - Introduction

1.1 Introduction

This thesis describes a project to design and implement a data abstraction facility for LISP. The result of this project, the CADRS (Compiler for Access of Data Regardless of Structure) system, supports the definition, creation, and efficient manipulation of hybrid data / procedure structures that closely resemble class objects in languages like Simula [2] and Smalltalk.

1.2 An Overview of this thesis

This thesis is intended to serve two purposes. This first is to document the design philosophies behind CADRS. The second is to serve as a reference to the features of CADRS for those who wish to use the package.

The remaining sections of this chapter describe the design of CADRS. This discussion begins with a brief inspection of the different data abstraction formalisms. Next a brief overview of the features of CADRS is presented. This is followed by a comparative analysis of CADRS and other LISP based structure definition packages. The chapter is concluded with a comparison of CADRS and FRL, a network-based knowledge representation scheme.

1. CADRS is currently implemented under Franz Lisp Opus 38, but could easily be modified to run under MACLisp.
The following chapters describe CADRS in detail. These chapters (2 through 11) are intended to serve as the reference manual for CADRS. Each of the user-level functions in the package is described and illustrated.

The final chapter presents some concluding remarks and considerations for future work.

1.3 An Overview of Data Abstraction Schemes

As pointed out by Worth in [9], "... a large amount of information that is to be processed [by a computer] in some sense represents an abstraction of a part of the real world." The first step in this processing is the development of a representation of this abstraction on the computer. As Worth points out, the lowest level of this representation (and the one to which all others are mapped), is that of the data and control structures offered by the hardware (or virtual machine for LISP). The data abstraction facilities found in high-level languages, then, provide more complex primitives, allowing the representation to be constructed on a more conceptual level.

1.3.1 Basic Data Abstraction

PASCAL provides the essentials necessary to support reasonable data abstraction. The first is the ability to define data types in terms of previously defined types. The second is its ability (through the use of RECORDS) to define heterogeneous data structures, allowing logical grouping of information. The third is its ability to define dynamic (recursive) structures.
3 July 1984

The structure definition scheme of PASCAL is not without its limitations. As an illustration, consider the following PASCAL program that implements a stack to reverse a list of numbers:
program reverse (input, output);
  type
    stack = ^stk;
    stk = record
      val : integer;
      nxt : stack
    end;
  var
    i : integer;
    s : stack;

  procedure push (v : integer; var s : stack);
    var
      tmp_s : stack;
    begin
      tmp_s := s;
      new (s);
      s^.val := v;
      s^.nxt := tmp_s
    end;

  function pop (var s : stack) : integer;
    begin
      pop := s^.val;
      s := s^.nxt
    end;

  function empty (s : stack) : boolean;
    begin
      empty := s = nil
    end;

begin
  s := nil;
  write ('enter value');
  while not eof do
    begin
      readln (i);
      push (i, s);
      write ('enter value');
    end;
  writeln ('the reversed list is');
  while not empty (s) do writeln(pop (s))
end.
The **push** and **pop** routines insulate the programmer from the underlying structure of the stack. Unfortunately, the ease with which the structure can be directly manipulated tempts the programmer to occasionally do so rather than write the necessary function / procedure (e.g. the use of ‘s := nil’, rather than some function/procedure to initialize the stack s). Thus a change to the underlying structure of the ‘stack’ has the potential to render invalid some of the procedures that use the structure. For instance, if we could place an upper bound on the number of entries in the stack, we might change the underlying structure to:

```plaintext
stack = record
    val : array [1..100] of integer;
    head : num : integer;
end;
```

Another limitation of the scheme is that it does not allow for the definition of a field item in terms of other items within the structure. For example, it would be desirable to define an employee as:

```plaintext
employee = record
    name : a name;
    age : 1..80;
    wage : real;
    hours : real;
    pay : (wage * hours);
end;
```

where **pay** is a read-only field that is computed each time it is accessed.
1.3.2 Object Oriented Data Abstraction.

SIMULA67 [2] allows the definition of a more robust structure, known as a **class**. Classes are defined in terms of data primitives, procedures, functions and other classes. The definition of a class resembles that of a procedure definition where the body of the procedure (i.e. the class body) is the code that will initialize a new **object** of that class. Assuming a PASCAL that was extended to understand CLASSes we could rewrite the stack example as:
program reverse (input,output);

class stack;
  type
    stk_p = ^stk;
    stk = record
      val : integer;
      nxt : stk_p
    end;
  var
    s : stk_p;

procedure push (v : integer);
  var
    tmp_s : stk_p;
  begin
    tmp_s := s;
    new (s);
    s^.val := v;
    s^.nxt := tmp_s
  end;

function pop : integer;
  begin
    pop := s^.val;
    s := s^.nxt
  end;

function empty:boolean;
  begin
    empty := s = nil
  end;

begin {stack initialization}
  s := nil
end; {stack}

var s : stack; i: integer;

begin
  new (s);
  write ('enter value ');
  while not eof do
  begin
    readln(i);
    s.push (i);
    write ('enter value ')
  end;
  writeln ('the reversed list is');
  while not s.empty do writeln(s.pop)
The concept of a stack is quite different in the two programs (albeit, they look quite similar). In the pure PASCAL program, we view push and pop as functions that manipulate the stack structure. In the object-oriented approach, one requests information from the stack (s.pop) and makes requests of the stack (s.push). It is this view of the object instance as a 'black-box' entity that firmly insulates the object's underlying structure from the program. In addition, some implementations of class/object structures [7] can selectively limit the 'visibility' of the object's components (e.g. data-structures or functions/procedures); a feature not found in the PASCAL like schemes.

1.4 A Short Overview of CADRS

CADRS realizes many of the advantages of an object-oriented system even though it is not truly object-oriented. This is accomplished through the extension of more conventional structure definition schemes. The following summary of CADRS should help illustrate this last point, as well as aid the reader during the comparative analysis that follows.

The structure function is used to define all CADRS structures. Its first argument is the name of the structure to be defined. The remaining arguments are taken to be a series of attribute/attribute-value pairs which serve to instantiate the structure definition.
The following is a list of the valid structure attributes and a description of each:

- **TYPE**

  The *type* attribute describes the underlying LISP structure that should be used to implement the structure being defined. The values allowed are:

  - list - for a list.
  - a-list - for an association list.
  - p-list - for a disembodied property list.
  - array - for a MACLisp style array.
  - hunk - for a hunk.
  - hash - for a hash link.

  If this attribute is not specified, then the value list is assumed.

- **FIELDS**

  The value of the *fields* attribute is the list of the data items to be defined for the structure.

- **TO-CREATE**

  The value of the *to-create* attribute is an expression which when evaluated will create an instance of the structure being defined. This is used to override the form computed by CADRS.

- **BEFORE-CREATING**

  The value of the BEFORE-CREATING attribute is a form to be evaluated before each instance of the structure is created.
o AFTER-CREATING

The after-creating attribute is used to specify a form to be evaluated after an instance of the structure is created. The atom datum is lambda bound to the created instance while the after-creating form is being evaluated.

o IS-A

The is-a attribute allows a structure to inherit the definition of one or more previously defined structures.

o TYPECHECK

The typecheck attribute is used to tell CADRS to typecheck all created instances of the structure.

o TYPE?

The value of the type? form is taken to be a lambda expression or the name of a function which, when evaluated against an arbitrary form, will return t if that form is an instance of the structure and nil otherwise.

Fields, like structures, have attributes whose values may be assigned during the definition of the structure. The valid list of field attributes are:

o IS-A

The is-a field attribute defines the field as an instance of the structure named as the value of the attribute.

o TO-FETCH

The to-fetch attribute is used to specify a lambda expression or 'pattern' that will return an expression to fetch the value of the field from a form. This is used to override the CADRS generated form.

1. This mechanism is shown in the CADRS reference chapters.
2. For example (car datum) if the fetch should return the first element in a list.
3 July 1984

- TO-REPLACE

The value of the **to-replace** form is a lambda expression or 'pattern' that will return a form to replace the field's value in a form. This is used to override the CADRS generated replace expression.

- DEFAULT

The value of the **default** attribute is an expression to be evaluated to give the field a default value. If this attribute is not specified, then the default value of the field will be the value of the symbol cadrsFieldDefault.

- TYPE

The **type** of a field defines whether the field actually takes up physical space in the underlying structure, or is defined solely in terms of function/procedure calls. If the value of **type** is **physical** then a cell in the underlying structure is reserved for it. If the value is **virtual** then no space is reserved.

- HASH-SIZE

The **hash-size** attribute is valid only for fields in a structure whose underlying type is **hash**. The value of the attribute is used to specify the size of the hash-table for this field.

- BEFORE-FETCHING

This attribute is used to specify a form to be evaluated before fetching the value of the field from a form. The symbol **datum** is lambda bound to the form the fetch is being performed on during the evaluation.

- AFTER-FETCHING

This attribute is used to specify the form to be evaluated after fetching the value of the field from a form. The symbols **datum** and **value** are lambda bound to the form the fetch is being performed on and the value of the field respectively.

- BEFORE-REPLACING

The value of this attribute corresponds to that of the **before-fetching** attribute for the replace operation. The symbols **datum** and **newvalue** are lambda bound to the obvious values.
3 July 1984

o AFTER-REPLACING

The value of the after-replacing attribute is a form to be evaluated after the value of the field is replaced. The symbols datum and newvalue are lambda bound to the obvious values.

o MUST-BE

The value of the must-be attribute is a form that is evaluated each time the value of field is replaced. During its evaluation, the symbols datum and newvalue are lambda bound to the obvious values. If the form returns t then the replace is performed. If nil is returned, then an error occurs¹.

The create function is used to create an instance of a structure. Among other things, create allows the user to initialize any of the fields or sub-fields of the structure.

The functions that provide access to the fields of a structure instance are fetch and replace. fetch returns the value of field based upon the data-path specified, and replace will replace the value of a field with a new value.

1.5 Comparative Analysis

This section seeks to compare CADRS to three of the more popular LISP based structure definition packages. First, CADRS is compared to the DEFSTRUCT package written by Alan Bawden [1]. Next it is compared to the INTERLISP record package written by L.M. Masinter [8]. Finally is compared to Gordon Novak's GLISP [4].

¹. Note that this error can be trapped.
1.5.1 DEFSTRUCT

The DEFSTRUCT package is a structure definition tool that runs under PDP10 and Multics MacLisp and on LispMachines. It has many of the capabilities of CADRS and is extendible.

Both of the packages provide a variety of choices for the underlying LISP structure to be used when an instances is created. The CADRS implementation of list structures is slightly more powerful since it allows the user to specify the exact format of the list (in terms of embedded sublists and using the last cdr of a cons cell to hold a value). The choices provided by DEFSTRUCT are a linear list or a balanced tree.
A drawback in the DEFSTRUCT scheme is that a field, or slot as it is called in DEFSTRUCT, can not be explicitly declared to be an instance of another structure. The difference is illustrated in the following example.

```
--CADRS definition of 'name' and 'person'
(structure name fields (last first . mi))
(structure person type array
   fields ((declare name is-a name) age sex))

(fetch a-person name first)

(create person
   (age . 24) (sex 'Male)
   ((name last) 'Duncan) ((name first) 'Richard)
   ((name mi) 'J))

--DEFSTRUCT definition of 'name' and 'person'
(defstruct (name (type list*))
   last first mi)
(defstruct (person (type array))
   name age sex)

(name (person a-person))

(make-person age 24 sex 'Male name '((Duncan Richard . J))
```

The access to the different fields/slots (and subfields/subslots) in the person structure are identical. The problem lies in the fact that DEFSTRUCT has no way of knowing that the name slot in the person structure is defined as a name structure. As a result, the slot must be initialized (with a list), rather than initializing each of the subslots. This is a potential source of problems if the underlying structure of the name structure were to change (to array, for instance).

While both packages allow one structure to include another's definition (the is-a structure attribute in CADRS and the include option in DEFSTRUCT), the DEFSTRUCT version is much less flexible. The include option can only be used on
structures with certain underlying types and only the slot initialization expression may be changed from their original values. In CADRS, any of the structure or field attributes may be changed or modified, and any structure type may be inherited.

The create function provided by CADRS supports the combined functionality of the make-x and alter-x macros defined by DEFSTRUCT\(^1\). In addition, create supports many other features (needed for compatibility with the INTERLISP Record Package) such as:

- **create** can be told to smash (like alter), use the top-level structures of, or copy the top-level structures of another structure in the creation of an instance.

- **create** preserves the order of evaluation of any field initializations provided. To illustrate this, consider the structure and create expression:

```
(structure test-init fields (a b c d e f))
(create test-init (e . 10) (f . 20) (d . 30)
  (b . 40) (c . 50) (a . 60))
```

The form that would be evaluated to satisfy the order of evaluation given would be:

```
((lambda ($$e $$f $$d $$b $$c)
  (list a $$b $$c $$d $$e $$f))
  10 20 30 40 50)
```

\(^1\) Where \(x\) is the name of one of the structures currently defined.
3 July 1984

A number of features in CADRS are not directly available in DEFSTRUCT, but could conceivably be implemented using DEFSTRUCT's extension capabilities. These features are enumerated below:

1. There is no way to validate a new slot value when a slot value is changed.

2. All slot names must be unique (i.e. no typing)

3. There is no procedural attachment for structure creation or slot access.

4. Virtual fields (slots) are not supported.

1.5.2 INTERLISP Record Package

Since the design of CADRS is based on that of the INTERLISP Record Package (ILRP), the two share much in common. There is a large intersection of underlying structure types that are supported by each\(^1\). The functions used for instance creation and structure access are, for the most part identical.

The ILRP can specify substructures local to the structure\(^2\) being defined -- a mechanism not supported in the current version of CADRS. For example, one could define a message as:

```
(record msg ((from . to) . mess) (record mess (header . body)))
```

This provides a means of accessing either the entire mess field or one of its

\(^1\) The disjoint sets are composed of those types that cannot be implemented in the host LISP.

\(^2\) ILRP structure are called records.
3 July 1984

components without having to define a separate mess record, as would be required in CADRS. In addition, the ILRP allows a field to have more than one such definition.

Each of the two packages support an instances typing mechanism, albeit the two differ greatly in how type is determined. CADRS explicitly types each object created using a hash-array and uses this feature to compute the type of a form\(^1\). The ILRP determines the type of a form by developing a predicate expression that checks the form against the record definition. Consider the following record declaration:

\[
\text{(record foo (a (b c (d)) e))}
\]

To determine if the value of \(x\) was an instance of a \texttt{foo}, the Record Package would evaluate a form similar to:

\[
\text{(and (listp x) (eq (length x) 3)}
\text{ (listp (cadr x) (listp (caddadr x)))}
\]

A CADRS feature that the ILRP does not share is the ability of a structure to inherit another structure's definition. This might be accomplished to some degree through the use of a user defined record type\(^2\).

---

1. Unless a \texttt{type?} form is supplied for the structure
2. The INTERLISP Record Package is somewhat extensible
The ILRP does not facilitate the types of procedural attachment supported by CADRS. Hence, the before/after creating/fetching/replacing procedural attachments could only be realized by the creation of a user defined record type or through the use of an ACCESSFNS record type\(^1\).

### 1.5.3 GLISP

GLISP is a strongly typed object oriented programming language which is built on top of LISP. The language uses an English-like syntax, and sports a compiler that enables an object based message facility and the ability to determine data path inference through the context of a computation [4]. Since CADRS is not intended as a programming language, but more as a tool, only the object/structure definition scheme will be studied for the purpose of comparison.

In general, CADRS compares favorably with the GLISP object/structure definition scheme. One major difference is the degree to which GLISP structures and objects are strongly typed (this is used by the compiler for inference of type in computation).

Both have the ability to build a heirarchy of structure types, and extend the definitions of a structure type\(^2\). Each has a sophisticated mechanism for procedural attachment, allowing the implementation of virtual fields.

---

1. ACCESSFNS records require that all of the structure access and creation code be supplied by the user.
2. \textit{is-a} in CADRS \textbf{transparent} in GLISP.
Some of the facets of the message facility of GLISP objects can be mimicked in CADRS using the **replace** function in conjunction with the **to-replace** field attribute. The major limitation would be that the messages could only have one argument. The operator overloading could not be duplicated without rewriting the necessary functions (in effect to perform the work of the GLISP compiler).

1.6 CADRS and Representation Schemes

This section compares CADRS and network-based schemes for knowledge representation as embodied in FRL [5]. The purpose of such a comparison is neither to insinuate that CADRS is, in any sense of the word, a knowledge representation tool, nor to propose that it be used as such. Rather, the goal is to characterize the descriptive capabilities of such knowledge representation schemes to gain insight into the kinds of capabilities that would be applicable to the description of data structures. The following short summary of the network based approach is presented to aid the reader during the comparison.

As stated by Mylopoulos in [3], network based knowledge representation schemes, "often called semantic networks, attempt to describe a world in terms of objects (nodes) and binary associations (labelled edges), the former denoting individuals and the latter binary relationships in the world being modelled." As such, the knowledge base can be viewed as a directed labelled graph in which the addition or deletion of new nodes/edges constitute the addition/deletion of knowledge about the world.
In FRL (Frame Representation Language), the description of an object, known as a frame, bears a close resemblance to that of a CADRS structure. The structure of a frame is:

[frame
  (slot1
    (facet1
      (datum1 (label1 message1 message2 ...)
        (label2 message1...))
      (datum2 (label1 ...))
    )
  (facet2 ..)
  (slot2 ..)
]

Frames can be related in a hierarchy as specified by the value of a frame's AKO (A Kind Of) slot (e.g. the value of the AKO slot for the frame DOG might be ANIMAL), and each slot may be specified as frame (e.g. the frame CAR that contains the slot WHEEL which is also a frame). Each slot can have a number of facets which serve to describe it. Facets can also serve to specify the invocation of procedures:

- when the value of the slot is changed or accessed.
- to supply the value of a slot that has no VALUE facet.
- to supply a default value for a slot with no VALUE facet.
- to validate the value of the slot.

A number of the ideas behind the structure of FRL frames and their associations have been adapted for use in CADRS. The notions of generalization (AKO in FRL and IS-A for CADRS structures) aggregation, the ability to define the parts of an object as other objects, and procedural attachment are useful in both types of description.
Also of use is the ability to classify object instances with respect to their type.

Many of the differences between FRL and CADRS are a result of the goals of each scheme. For example, the need to add new slots to a frame as knowledge of the object grows, has no counterpart in the structure definition domain since structures are typically static. Hence, CADRS does not provide a means to add new fields to an existing structure. Moreover, the type of inheritance supported by a structure definition scheme need not be as dynamic as that of a knowledge representation; again, due to the assumption that structure definitions are static.

As illustrated above, each slot in an FRL frame can contain user defined facets in addition to those understood by the system. While CADRS attributes are restricted to those that are system defined, relaxing this constraint to allow information to be associated with a structure/field seems very worthwhile and is under consideration.
2 - CADRS Reference

2.1 Introduction

The next 10 chapters are devoted to an in depth treatment of the various features of CADRS. As mentioned, these chapters are intended to serve as a reference manual for version 1.00 of the CADRS package and its implementation under Franz Lisp Opus 37.
The **structure** function is used to define CADRS structures as well as to describe their interrelationships. As in INTERLISP (or PASCAL for that matter), **structure** does not create instances of the structure being defined. As such, no relevant value is returned. As with much of the CADRS top level, **structure** is implemented as a macro, which in turn uses lower level functions to actually define the structure.

### 3.1 Syntax

The syntax for **structure** is:

```
(structure attrib1 [mod1] val1 attrib2 [mod2] val2 . . .)
```

where:

- **attribi** is one of the exceptible structure attributes (see below).

- **modi** is the optional modifier on that attribute.

- **vali** is the value to be associated with that attribute.

All of the attributes are optional.

---

1. Some attributes take on modifiers that specify how their values will be inherited. Attribute modifiers are covered in the chapter on Structural Inheritance.
3 July 1984

3.2 Structure Attributes

Although all structure attributes (and field attributes for that matter) have default values, and thus, need not be specified, the actual value that the attributes will take on depend on the "lineage"\(^1\) of the structure. While this point is covered more completely in the section on structure inheritance, it should be pointed out here that the default values mentioned below are in fact affected by the lineage of the structure.

3.2.1 type

The value of the type attribute specifies the underlying LISP structure that will be used to create an instance of this structure. The allowable values are:

1. list (The default) The fields in a structure of type list are stored as elements in a list.

2. a-list The value for a field is stored as the value of key <field> (whatever the name of the field is) in an association list. For example, if a structure has the field foo with the value bar and the field dog with the value cat, the a-list might look like:

   ((foo . bar) (dog . cat) . . .)

3. p-list Values are stored as the value of indicator <field> in a disembodied property list\(^2\). If the fields and values above were in a structure of type p-list, it might look like:

---

1. The lineage of a structure is the set of structures that have been inherited, along with the lineage of each of the respective structures in that set.
2. A disembodied property list is one which is not accessed (referenced) as the property list sub-part of a symbol.
3 July 1984

(foo bar dog cat . . )

4. **array** The value of a field is stored in the array element which has been mapped to that field. Using the same example, we might have an array which looks like:

```
+-------+
|   bar  |
+-------+
|   cat  |
+-------+
```

5. **hunk** The value of a field is stored in the hunk element which has been mapped to that field. Using the Franz notation for hunks we might have something like:

```
{bar cat}
```

6. **hash** Hash structures are quite different from the other structures supported by CADRS\(^1\). A major difference is that field values are stored (hashed) in a separate structure in which field values are accessed using the `form` itself as the key. Thus, once a hash structure is defined, any LISP structure (lists, symbols, arrays, CADRS structures etc), 'have' the field mentioned in the hash structure. Another difference is that hash structures can not be created (using the `create` function), since the 'creation' is done when the structure is defined.

One of the most significant uses of hash structures is to extend existing structures. The following example illustrates both hash structures and the technique for extending structures\(^2\).

---

1. The Franz version of CADRS provides the necessary functions to support the **hash** type. Hash structures are covered in detail below.
2. Since this example does use functions that are described later in the manual, it might be worth your while to skip it for now, and come back to it later.
Consider the following structure:

(structure some-data type hunk
  fields ((declare f1 is-a foo)
    (declare f2 is-a bar)
    f3
    f4
    (declare f5 is-a foobar)
  ))

If we had a large network built using instances of some-data and then discovered a need to add field f6, the only recourse (without using hash structures) would be to unload all of the items and start from scratch\(^1\).

If we define a hash structure:

(structure add-the-field type hash
  fields ((declare f6 default nil)))

and then redefine the some-data structure as:

(structure some-data type hunk
  fields ((declare f1 is-a foo)
    (declare f2 is-a bar)
    f3
    f4
    (declare f5 is-a foobar)
    (declare link is-a add-the-field)
  ))

then all of existing some-data instances will now have a field f6 since for any some-data instance sd:

(replace sd link f6 100) ==> (puthash sd 100 f6)

(f6 is the hasharray used).

\(^1\) Even if some-data were a list, it would be necessary to 'cons' on the extra space for the field.
3.2.2 **fields**

The **fields** attribute defines the different data items that comprise the structure being defined. The default for the **fields** attribute is `nil` which means 'no fields'.

The value of fields, called the **field-list**, is a list of the fields in the structure. This list serves two purposes. The first, obviously, is to define the fields. The second is to define the position of the field within any structure of type **list**, **array**, or **hunk**. For **list** structures, this list can contain sublists which serve to define the embedded lists within the toplevel structure. As with structure definitions, each of the fields have attributes with associated modifiers and values. The specification of these are discussed in the following chapter.

3.2.2.1 **examples**

The following examples show how the different field-lists are handled in different structures.

1. (**structure name type list fields** (last first mi))
   
   If foo is an instance of a **name** structure then:

   
   (car foo) => last
   (cadr foo) => first
   (caddr foo) => mi

1. For compatibility, embedded field-lists are allowed for all types. For types other than **list**, the field list is processed as if it were linear.
2. If the type of the name structure (in the previous example) were a-list then:

\[
\begin{align*}
(cdr (assq 'last foo)) & \Rightarrow last \\
(cdr (assq 'first foo)) & \Rightarrow first
\end{align*}
\]

3. If name was of type hunk then:

\[
\begin{align*}
(cxr 0 foo) & \Rightarrow last \\
(cxr 1 foo) & \Rightarrow first
\end{align*}
\]

4. (structure bar type list fields ((a b) ((c) d) . e))
   If foo is an instance of a bar then:

\[
\begin{align*}
(caar foo) & \Rightarrow a \\
(caaadr foo) & \Rightarrow c \\
(cddr foo) & \Rightarrow e
\end{align*}
\]

5. If bar were of type array then:

\[
\begin{align*}
(funcall foo 0) & \Rightarrow a \\
(funcall foo 3) & \Rightarrow c \\
(funcall foo 5) & \Rightarrow e
\end{align*}
\]

3.2.3 create related attributes

The following attributes control what happens when an instance of the structure is created.
3.2.3.1 to-create

The to-create attribute is used to "advise" the create function of how to create an instance of the structure being defined. The value of the attribute is a form which, when evaluated, returns an instance of the structure. By default, the structure function builds such a form which, in most instances, will be at least as efficient as the user-defined form.

The following example shows how the to-create attribute is used:

(structure name type list
  fields (last first mi)
  to-create (list nil nil nil))

When create is called to create a name, it will evaluate the expression (list nil nil nil), which returns (nil nil nil).

3.2.3.2 before-creating

The value of the before-creating attribute is a form which is to be evaluated before an instance of the structure is created. By default, no such form is generated.

If before-creating is used as in:

(structure foo
  fields (a b c)
  before-creating (bar))

then create will return the result of a form similar to:
where <create-form> is the form that will create the instance.

3.2.3.3 after-creating

The after-creating attribute specifies a form to be evaluated after the instance of the structure is created. By default, no such form is generated.

The created instance is bound to the atom datum which may be referenced in the form. This feature can be useful in tracking the instance, and/or in modifying the instance after it is created. For example, when a structure such as:

```
(structure foo
  fields (a b c)
  after-creating (progn
    (print datum)
    (terpri))
```

is created, create will evaluate a form similar to:

```
(let ((datum (<create-form>)))
  (progn
    (print datum)
    (terpri)))
```
3.2.4 Structure typing

The following attributes control how CADRS determines the type of a structure instance.

3.2.4.1 type?

The type? attribute is used to tell CADRS how to identify a LISP entity as being an instance of the structure being defined. Although CADRS can do structure typing, the feature only works with instances created using the create function. The type? attribute provides greater control over how 'typing' is accomplished.

The value of the type? attribute is a lambda-expression or function of one argument which returns t if the argument is an instance of the structure being defined, and nil otherwise. The following is an example of the use of the type? attribute.

(structure name type list
  fields (last first mi)
  type? name-type?)

(defun name-type? (struct)
  (and (dtpr struct)
       (eq (length struct) 3)))

3.2.4.2 typecheck

The typecheck attribute is used to direct CADRS to typecheck all instances of the structure. By default, typechecking is only turned on for a structure when it contains a field whose definition conflicts with that of a field by the same name in
3 July 1984

another structure. If the value of `typecheck` is `t`, then type checking will be done by CADRS for all instances of the structure.

CADRS generated typing and a user supplied typing form (value of `type?`) can both be enabled for a structure. The reason for doing this is to allow typing to occur on more than one level. This concept is explained in greater detail in the chapters that follow.

3.2.5 `is-a`

The `is-a` attribute allows the structure to inherit the definitions of one or more previously defined structures. This attribute is restricted in that it must appear as the first attribute used in the call to `structure`.

If the value of the `is-a` attribute is an atom, the structure will inherit the definition of the structure named by the atom. If the value is a list, then each of the atoms in the list are taken to be previously defined structures whose definitions will be inherited in the order that they were specified.

The effect of `is-a` is covered completely in the chapter on Structural Inheritance.
4 - Field Definition

CADRS provides a high degree of control over the field level definition of the
structure using the same basic attribute/attribute value syntax as used on the
structure wide level. This chapter provides the details of field declaration
specific to the field-list (the value of the field attribute) and the mechanism for
specifying field attribute values.

4.1 Field-list specification

In general a field-list is any valid list in which the following is true:

- Each item in the list (a field) is unique within that list (i.e. within that
  structure).

- The list is acyclic (i.e. it terminates).

The following are examples of valid field-lists:

1. (a b c)

2. (a . b)

3. (a (nil nil) b)

The first example shows a typical field-list. In the second example, field a would
be mapped to the car of the list (assuming of course that the structure was of type
list), and field b would be mapped to the cdr of the list. In the third example, the
nils are used to reserve space in the list (i.e. unnamed fields similar to a fill
in COBOL).
4.2 Syntax

As mentioned, fields, like structures have attributes which can be modified within the structure definition. When a field is to take on the default values for its attributes, only the name of the field need appear in the field-list as in:

(structure foo fields (a b c))

To specify a value for a field attribute, the name of the field is replaced in the field-list by:

(declare field-name
      attr1 [mod1] val1 ...)

where:

- **field-name** is the name of the field to be defined.
- **attri** is a valid field attribute.
- **modi** is the modifier for that attribute.\(^1\)

Since none of the field attributes need be specified, the field-list

((declare a) ((declare b) (declare c)) declare d)

is equivalent to

---

\(^1\) As with structure attributes, some field attributes can take on modifiers to specify how these attributes should be inherited. Field attribute modifiers are covered in the chapter on Structure Inheritance.
4.3 field Attributes

When an attribute is not specified in a field definition, the actual value that the attributes will take on depends on the lineage of a structure. While this point is covered more completely in the chapter on Structure Inheritance, it should be pointed out here that the default values mentioned below are in fact affected by the lineage of the structure.

4.3.1 is-a

The is-a field attribute defines a field to be an instance of the structure named by the value of the attribute. This provides a handy structure building mechanism as shown below:

(structure name fields (last first . mi))

(structure person
  fields ((declare name is-a name)
    sex
    age))

Here, the first field in the any person structure, will be of type name and will have the fields last first and mi accordingly. An instance of a person structure might look like:
Note that the is-a field attribute has no meaning for structures of type hash and is ignored for these structures.

4.3.2 default

The value of the default field attribute is taken to be an expression to use as the default value of the field (i.e. the value given to the field when an instance of the structure is created). By default (no pun intended), the value of default is the atom cadrsFieldDefault which is initialized by CADRS to nil.

The default attribute is ignored for hash structures under version 1.00 of CADRS.

4.3.3 fetch and replace definition

The CADRS functions that are used to return the value of a field and to put a value in a field are fetch and replace respectively. The following field attributes control how CADRS performs field access, and 'customization' of this process. By default, CADRS computes the access and storage function for each field in the structure.

1. These functions are often used in conjunction with the to-fetch and to-replace field attributes described below to enhance the process.
4.3.3.1 to-fetch

The value of the to-fetch field attribute can be one of two things. If it is a lambda expression, CADRS assumes that when this is evaluated against the structure instance named in the fetch, it will return the proper expression to fetch the field, much like the body of a macro. By default, CADRS generates this form. As an example, consider the structure definition:

(structure a-structure
  fields ((declare a
to-fetch (lambda (form) (list 'car form)))))

If fetch was called to fetch the a field of the structure instance s as in:

(fetch s a)

the lambda would be evaluated with the argument s and return:

(car s)

If the expression is not a lambda, CADRS assumes that is to be used to 'show' CADRS how to fetch the field from a structure instance. For example, consider the structure:

(structure a-structure
  fields ((declare a to-fetch (cdr datum)))))

The atom datum in the value of the to-fetch field attribute directs CADRS to insert
the structure instance being fetched at this point. Thus if the fetch

    (fetch s a)

is evaluated, CADRS will evaluate the expression

    (cdr s)

to perform the fetch.

4.3.3.2 to-replace

The to-replace field attribute corresponds to the to-fetch for storing a value in a field. Like to-fetch, the value of the attribute can either be a lambda or an expression. In this case, however, the lambda must accept two arguments; the first being the form to do the replace on and the second the value to replace the old value with. The following example shows how this is done:

    (structure struct
      fields ((declare a
        to-fetch (lambda (form val)
          (list 'rplacd form val)))))

    (replace a-struct-instance a 'this-new-value)

    --will evaluate as--

    (rplacd a-struct-instance 'this-new-value)
As with to-fetch, if the expression is not a lambda, CADRS assumes that it is to be used to 'show' CADRS how to replace the value of the field from a structure instance. As before, if CADRS finds the atom, datum, in the expression, it will be substituted with the instance of the structure that the replace is being performed on. Similarly, if CADRS finds the atom newvalue in the expression, it is substituted with the value specified in the call to replace. The following example shows how this occurs.

(structure struct
   fields ((declare a
            to-fetch (rplaca datum newvalue))))

(replace asi a 10) => (rplaca asi 10)

4.3.3.3 before-fetching

The value of the before-fetching field attribute is a form to be evaluated before a fetch is performed on the field. By default no such form is evaluated. CADRS will evaluate the expression within a context in which the structure instance is lambda-bound to datum. An example of this is shown below:
(structure struct
   fields ((declare a
           before-fetching (note-fetch 'a datum)))

(fetch asi a)
   --will evaulate as--

(let ((datum asi))
   (note-fetch 'a datum)
   (car datum))

In the example, **note-fetch** might be considered as some function that would record the instance of the fetch.

4.3.3.4 **before-replacing**

The value of the **before-replacing** field attribute is a form to be evaluated before a replace is performed on the field. By default no such form is evaluated. The expression is evaluated within a context in which the structure instance and the value that is to replace the current field value are lambda-bound to **datum** and **newvalue** respectively. An example of this is shown below.
3 July 1984

(structure struct
  fields ((declare a
    before-replacing
    (note-replace
      'a
      datum
      newvalue)))))

(replace asi a 10)

--will evalutate as--

(let ((datum asi) (newvalue 10))
  (note-replace 'a datum newvalue)
  (rplaca datum newvalue))

In the example, note-replace might be considered as some function that would record the instance of the replace.

4.3.3.5 after-fetching

The value of the after-fetching field attribute is an expression to be evaluated after the value is fetched from the form. By default, the fetch form is the last expression evaluated. CADRS will lambda-bind the structure instance that the fetch is being performed on and the value fetched to the datum and value respectively as shown below:

(structure str
  fields ((declare a
    after-fetching (foo datum value)))))

(fetch a-str a) ==> (let*
  ((datum a-str)
   (value (car datum)))
  (foo datum value)
  value)
4.3.3.6 after-replacing

The value of the after-replacing field attribute is an expression to be evaluated after the field value is replaced in the form. By default, the replace form is the last expression evaluated. CADRS will lambda-bind datum and newvalue to the obvious values as shown below:

(structure str
  fields ((declare a
    after-replacing (foo datum newvalue))))

(replace a-str a 10) ==> (let
  ((datum a-str)
    (newvalue 10)
    (%%ret (rplaca datum newvalue)))
  (foo datum newvalue)
  %%ret)

4.3.4 type

The value of the type attribute dictates whether or not the field will have storage allocated for it within the structure. If the value of type is physical (the default), then a storage cell, (e.g. a list cell in a list structure), will be allocated for the field. If the value is virtual then no storage is allocated.
3 July 1984

This feature is useful when a field is defined in terms of other fields within the structure as in:

(structure employee
   fields ((declare name is-a name)
   wage
   hours-worked
   (declare pay
      type virtual
      to-fetch
      (* (val-of datum wage)
         (val-of datum hours-worked)))))

If a instance of an employee, as defined above, has a wage of 10 and an hours-worked of 20 then the call:

(fetch an-employee pay)

would return the value 200.

It is important to realize that a virtual field must have a to-fetch specified to perform the fetch on this field. Similarly a to-replace form must be specified to perform field replacement¹.

¹ The val-of and val-to functions can not be used for virtual fields since no physical storage is allocated for them (and therefore CADRS can not generate the necessary functions).
4.3.4.1 must-be

The must-be field attribute provides the means to check field values for validity. By default, no validity checking is done by CADRS.

The value of the attribute is an expression which is to be evaluated each time a replace is done for the field in this structure. The structure instance and the new value are lambda-bound to datum and newvalue when the expression is evaluated. If the result of the evaluation is non-nil, the replace is performed. Otherwise control is transferred to the CADRS must-be-handler function. An example of a field definition with a must-be attribute is shown below.

(structure student
  fields ((declare name is-a name)
         (declare grade
           must-be
           (and (> 0 newvalue)
                (<= 100 newvalue)))))

(replace a-student grade 99)

evaluates as

(rplaca (cdr a-student)
  (let ((newvalue 99))
    (cond ((and (> 0 newvalue)
                 (< 100 newvalue))
           newvalue)
          (t (must-be-handler
              grade
              newvalue
              '(student)
              a-student))))))

1. This function and other error handling functions are explained in the chapter on Exception Handling.
4.3.5 hash-size

The hash-size field attribute specifies the size of the hash array CADRS will allocate for this field. This field attribute is meaningful only for fields in structures of type hash and is ignored otherwise.
5 - **create** Structure Instance Creation

The `create` function is used to create an instance of a structure. It has a syntax similar to the INTERLISP `create` function, maintaining the same flexibility, and including the following features:

- The ability to provide an initial value expression for all fields and subfields.
- The ability to create a copy of another structure instance.
- The ability to use a current structure instance in the creation of the new instance.
- The ability to 'reinitialize' a current structure instance.

Aside from the above, another reason for using `create` is that it allows the base type of a structure to change (e.g. from a list to a hunk), with no change to any of the functions that use these structures.¹

5.1 Syntax

The syntax for `create` is:

---

¹. Obviously, if you don’t use the CADRS fetch, replace and create functions and/or write structure dependant to-fetch/to-replace expressions, this will not be true.
3 July 1984

(create structure
   [cr-attrib struc-instance]
   [(field-name1 . expr1)]
   [(field-name2 . expr2)]
   .
   [(field-namen . exprn)])

where:

- **structure** - Is the name of the structure instance to create.

- **cr-attrib** - or create attribute, is one of using reusing copying or smashing. Unlike structure or field attributes, attributes for create are mutually exclusive. That is, a create with the using attribute, cannot have the copying attribute etc.

  The value of a create attribute is a structure instance to be used in the creation process. Create attributes are discussed below.

- **field-namen** - Is the name of a field to be initialized to the value of exprn. Field initialization is discussed below.

5.2 Notes on Structure Creation

The following should be kept in mind when structures are created:

1. The structure being created and the structure type of the instance 'used' must be the same. This is a limitation of version 1.00.

2. When a field is given an initial value, it is just as if the default attribute for that field is the value specified. Therefore, if the field has either to-fetch, before-fetching, after-fetching, or must-be attributes, their effects are ignored at this point.
5.3 Field Initialization

When create encounters a (field . expr) pair, the value of the expression is used as the initial value for the field. Consider the following structure definition and create call:

(structure team
  fields (name town coach standing))

(create team
  (name . 'Eagles)
  (town . 'Philadelphia)
  (coach . 'Vermeil)
  (standing . 1))

The form that create will actually evaluate is:

(list 'Eagles 'Philadelphia 'Vermeil 1)

CADRS pays attention to the order in which field evaluations are specified so that side effects from the evaluation of any of the expressions will produce the 'desired' effect. Consider the following example:

(let ((na 'Cowboys) (to nil)
       (co 'Landry) (st 2))
  (create team
    (name . na)
    (coach . (prog2
      (setq to 'Dallas)
      co))
    (town . to)
    (standing . st)))
This time, create will evaluate as:

\[
((\lambda (\$coach)
   (\text{list na to } \$coach \text{ st}))
\text{prog2}
   (\text{setq to 'Dallas})
   \text{co}))
\]

To initialize the fields of an embedded structure the path is specified by a list as the field name (rather than an atom). As an example, consider the following structures:

\[
(\text{structure name fields (last first . mi)})
\]
\[
(\text{structure person fields ((declare name is-a name))})
\]

To initialize the last field of a person structure, one would use the create expression:

\[
(\text{create person ((name last) . 'Smith})
\]
5.4 create attributes

In illustrating the differences between the `using` `reusing` `copying` or `smashing` attributes, we will consider the structure:

```
(structure foo
    fields (a b (c d) e))
```

with instance:

```
foo_inst <=>(10 15 (20 25) 30)
```

5.4.1 using

The `using` attribute will cause `create` to use the top level structures of the instance specified to create the new instance. Thus, in a `create` such as:

```
(create foo using foo-inst (a . 100) (c . 200))
```

`create` would generate

```
(list 100
    (cadr foo-inst)
    (list 200 (cadaddr foo-inst))
    (cadddr foo-inst))
```
3 July 1984

If the structure has any substructures such as:

(structure person fields ((declare name is-a name)))

create using will copy (similar to create copy) the substructures into the structure instance.

5.4.2 reusing

The reusing attribute causes create to use as much of the instance provided as possible in the creation of the new instance\(^1\). The create:

(create foo reusing foo-inst (b . 150))

would evaluate as

(cons
 (car foo-inst)
 (cons 150
  (cddr foo-inst)))

\(^1\) reusing will not be supported in V1.00 of CADRS. If reusing is specified, the structure will be created as a using.
3 July 1984

5.4.3 **copying**

The copying attribute directs create to copy the the top-level items from the instance using copy function. Thus:

```
(create foo copying foo-inst (d . 250))
```

will evaluate as

```
(list (copy (car foo-inst))
 (copy (cadr foo-inst))
 (list (copy (caaddr foo-inst))
   250)
 (copy (cadddr foo-inst)))
```

5.4.4 **smashing**

The smashing attribute causes create to smash the structure to perform the creation as shown:

```
(create foo smashing foo-inst (a . 100) (b . 150))

(let ((f foo-inst))
 (rplaca foo-inst 100)
 (rplaca (cdr foo-inst 150))
 f)
```
6 - Structure Access Functions

Described in this chapter are the \texttt{fetch/val-of} and \texttt{replace/val-to} functions which perform all user level structure access in CADRS. As is the case with the structure and create functions, all of the structure access functions are implemented as macros which can, in the proper mode replace themselves with the code to evaluate.

6.1 \texttt{fetch} and \texttt{val-of}

The \texttt{fetch} and \texttt{val-of} functions return the value of a field within a structure instance. The syntax for each is:

\begin{verbatim}
(fetch struc-inst path-spec)
(val-of struc-inst path-spec)
\end{verbatim}

where:

- \texttt{struc-inst} is the instance of the structure to perform the fetch on.
- \texttt{path-spec} is a list of one or more field names which explicitly determine a unique field within the structure.
3 July 1984

The following example demonstrates use of fetch to retrieve the value of a field from a 'simple' structure, and then from an nested structure.

(structure name fields (last first . mi))

(structure person
  fields ((declare name is-a name)
    sex
    age
    ss-number))

(fetch a-name first) => (cadr a-name)

(fetch a-person name) => (car a-person)

(fetch a-person name first) => (cadar a-person)

The difference between the two functions lies in the manner in which they deal with the value of the to-fetch attribute for the field. If a to-fetch form is specified for a field, the fetch function will use it rather than the CADRS generated form which would be used otherwise. The val-of function, on the other hand, uses the CADRS generated form regardless of the value of to-fetch, effectively overriding the to-fetch form\(^1\).

The reason for the distinction is to allow to-fetch forms to be written without using (structure dependant) pure LISP. Using val-of, one can easily write to-fetch forms which either filter the actual value of the field, or define the value to fetch based on other fields (for fields of type virtual) independant from the

\(^1\) As mentioned before using val-of to return the value of a field of type virtual will cause an error.
3 July 1984

underlying LISP structure used.

For example, the following shows an structure which could be used to store hours worked and wages for an employee.

(structure employee
   fields ((declare name is-a name) ; from above
          wage
          hours-worked
          (declare pay
              type virtual
              to-fetch
              (* (val-of datum wage)
                  (val-of datum hours-worked)))
       ))

6.2 replace/val-to

_replace_ and _val-to_ correspond to _fetch_ and _val-of_ for storing information in a structure. The syntax for each is:

(replace _struc-inst_ path-spec new-val)

(val-to _struc-inst_ path-spec new-val)

where:

- _struc-inst_ is the structure instance to be modified.

- _path-spec_ is a list of one or more field names which explicitly determine a unique field within the structure.
new-val is the new value for the field.

As with fetch and val-of, replace will use the to-replace form if one is present, while val-to will always use the CADRS generated form[^1]. The following example shows a method for defining a structure which increments its value each time it is fetched.

```
(structure counter
  fields ((declare count
to-fetch
  (let
    ((v (1+ (val-of datum count))))
    (val-to datum count v)
    v))
  ))
(setq c (create counter))
(replace c counter 10)
(fetch c count) => 11
(fetch c count) => 12
```

[^1]: As with val-to using val-of to replace a virtual field is an error.
The **type** and **type?** functions provide information about the type of a structure instance. While the more interesting applications of each of these is covered in the next chapter on Structural Inheritance, the functionality of each is covered here.

### 7.1 **type**

The syntax of **type** is:

```
(type struc-inst)
```

where:

- `struc-inst` is an instance of some structure.

The value returned by **type** is a list of one or more structures that `struc-inst` is an instance of. For example:

```
(structure sl
    fields (a b c)
    typecheck t)

(type (create sl)) => (sl)

(structure s2 is-a sl)

(type (create s2)) => (s2 sl)
```

In order for **type** to be used, typechecking must be turned on for the structure.

---

1. The different methods for doing this are covered in the chapter on Modes of Operation.
Otherwise, type will return the LISP type of the argument.

7.2 type?

The type? is a predicate function that returns t if its first argument is an instance of one or more of the structures in the second argument and nil otherwise.

The syntax for type? is:

(type? struc-inst struc-list)

where:

- struc-inst is an instance of some structure.
- struc-list is one of:
  - a previously defined structure.
  - a list of previously defined structures.

In order for type? to work correctly, the one of the following must be true about each member of struc-list (or the single structure):

1. Typchecking must be implicitly or explicitly turned on for the structure.
2. A type? form must have been specified (using the type? structure attribute.

If neither of these conditions is met, the type? exception handler will be called.

If a type? form is defined for the structure this is used in place of the CADRS

---

1. This is what type normally returns in Franz Lisp.
2. The type? exception handler is covered in the chapter on Exception Handlers.
7.3 typing in general

The structure typing performed by CADRS is straightforward. However, the mechanisms to alter typing (the type? structure attribute and function) provide a means to implement a much more sophisticated notion of type.

For example, given the definition of a person structure as:

```scheme
(structure person
    fields ((declare name is-a name)
            age
            sex))
```

We could define a concept of women as:

```scheme
(structure women
    type?
    (lambda (s)
        (and (memq 'person (type s))
            (eq (fetch s sex) 'female))))
```

and find that:

```scheme
(type? (create person (sex 'female)) (women))
```

would in fact, return t.
Further, it is possible, using CADRS, to define a global type handler to handle all requests to `type?`, thus providing additional granularity to instance typing. This technique is presented in the chapter on Exception Handling.
8 - Structural Inheritance

As a preface to the following material, the manner of Structural Inheritance presented here is not of the type found in the knowledge representations languages (FRL, KRL, KL-ONE). This is probably obvious considering that CADRS attributes are for the most part directive not descriptive and that the set of attributes is fixed. On the positive side, Structural Inheritance, does go beyond the features offered in other structure definition tools (INTERLISP Record Package, or MacLISP's DefStruct) in providing a means of extending and modifying structure definitions to create new definitions which are semantically related to the parent. As an example of the utility of such a feature, consider the following.
Note the following structures to deal with student and employee information:

(structure student
  fields ((declare name is-a name)
            (declare address is-a address)
              ss-number
              sex
              age
              school
              major
              courses
              advisor
              date-of-grad))

(structure employee
  fields ((declare name is-a name)
            (declare address is-a address)
              ss-number
              sex
              age
              manager
              division
              group
              wage
              hours
              ytd-hours))
It is apparent that there is a kernel of information which is common to both. In a conceptual sense this is, of course, due to the fact that both students and employees are people. To capture this relationship in CADRS, one could define student and employee in terms of a definition of a person such as:

(structure person
   fields ((declare name is-a name)
            (declare address is-a address)
            ssn-number
            sex
            age))

(structure student is-a person
   fields (school
            major
            courses
            advisor
            date-of-grad))

(structure employee is-a person
   fields (manager
            division
            group
            wage
            hours
            ytd-hours))

Both methods (with and without 'person') provide the same structure access (in terms of fetch or replace), and in this sense, the inheritance mechanism merely provides a convenient shorthand notation. In terms of instance typing, the second methods allows the user to check the type of person/employee instances at two levels of granularity (e.g. if x is a person, or if x is a student).

The first half of this chapter describes the method by which structure and field attributes are inherited. This is followed by a discussion of the semantic notion of inheritance with respect to instance type.
8.1 Attribute Inheritance

When a structure definition contains the is-a attribute, the attribute values of the structure specified become the default of the structure being defined. Thus if we have 2 structures s1 and s2 defined as:

(structure s1 fields (a b c))
(structure s2 is-a s1)

s2 by default will have the fields a b c, as would be expected. If s1 and s2 have the slightly different definitions:

(structure s1
   fields (a b c)
   after-creating (s1AfCr datum))
(structure s2
   is-a s1
   after-creating (s2AfCr datum))

the outcome is not quite so obvious since there are several possibilities all of which could be desirable in one situation or another. To deal with this, some attributes have modifiers which direct how their values are inherited.

8.1.1 Attribute Modifiers

The attribute modifiers in order of decreasing restriction are only, always and also. The effect of these on the value of an attribute depends on the modifier that was specified on the parent level. The following describes each modifier and its effects:
1. only. The only modifier prohibits modification of an attribute's value by any descendant of the structure. only may be used in a descendant provided that the attribute in the parent structure is modified with also (i.e. if the parent attribute is modified with either only or always an error will result). If only is used in a descendant structure, the inherited attribute value is overridden by the value specified and the modifier at this level becomes only.

2. always. This is the second most restrictive modifier. always can only be modified by the the also modifier. When always is inherited and modified by also, the current value is concatenated\(^1\). When also is inherited and modified by always, the values are concatenated and the modifier becomes always.

3. also. This is the least restrictive modifier. When the modifier is not specified for a modifiable attribute, also is the default used. When also is inherited, the value is overridden by only and concatenated for also and always. The modifier will become the one specified by the descendant structure. When also is specified as a descendant to always and also the values are concatenated and the modifier remains unchanged.

Finally, while a modifier need not be specified for a modifiable attribute, an error will result if a modifier is given for an attribute that is not modifiable.

Note as well that unmodifiable attributes are implicitly only.

\(^1\) The way in which the values are concatenated or grouped together depends on the attribute involved. This is explained below.
The following diagram shows the allowable modifier combinations along with the resulting values and modifiers.

```
  Descendant
      +---------------------+
      |        only        |
      |    always    also   |
      +---------------------+
P  | only     ERROR      ERROR      ERROR |
   a  |---------------------|
   r  | always      ERROR      ERROR      [val: concat]|
   e  |            [mod: always]|
   n  | [val: override] [val: concat] [val: concat]|
   t  | [mod: only]      [mod: always] [mod: also]|
```

8.1.2 Structure Attributes

The following describes how structure attributes are inherited. Each of the structure attributes are presented along with a description of how its value is inherited and how the different modifiers (if any are allowed) affect this process.

8.1.2.1 type

In version 1.00 of CADRS, type is an unmodifiable attribute, and thus its value may not be changed. This restriction could possibly be relaxed in a future version.
8.1.2.2 fields

The value of the fields attribute may be changed by a descendant, but the changes that are possible are somewhat restricted and therefore do not fall into the 'normal' framework. The way in which the values of the parent and descendant are concatenated is illustrated by the following example.

Given the 2 structures:

(structure s1 fields (a b c))
(structure s2 fields (d e f))

the concatenation of the field-lists has the effect of an order preserving union operation on the two values. The value resulting is:

(a b c d e f)

If one or more of the fields appear in both fields-lists, the values of the field attributes specified by the descendant field are used to modify the field attribute values of the parent. This operation is similar to structure attribute inheritance and is covered in the section on field attributes below.
3 July 1984

8.1.2.3 before-creating and after-creating

The **before-creating** and **after-creating** attributes both use the standard modifier syntax in their inheritance. The method of concatenation is to queue the forms and to insert them in order in the create expression as shown:

```
(structure s1
  fields (a b c)
  before-creating (foo)
  after-creating (bar))

(structure s2 is-a s1
  before-creating (s2foo)
  after-creating (s2bar datum))

(create s2) => (progn
  (foo)
  (s2foo)
  (let ((datum (list nil nil nil)))
    (bar)
    (s2bar datum)
    datum))
```

8.1.3 to-create type? and typecheck

The **to-create type?** and **typecheck** attributes are unmodifiable.

8.1.4 Field Attributes

Field attribute inheritance is implemented in much the same way as inheritance of structure attributes. The specifics of inheritance for each field attribute are presented below.
3 July 1984

8.1.4.1 default is-a to-fetch to-replace type

The \texttt{default is-a to-fetch to-replace} and \texttt{type} field attributes are unmodifiable and an error will result if a descendant tries to change the inherited values.

8.1.4.2 before/after fetching/replacing

\texttt{before-fetching, after-fetching, before-replacing, and after-replacing} are modifiable using the standard syntax. The method of concatenation is to queue the forms and to insert them in order in the fetch/replace expression as shown:

\begin{verbatim}
(structure s1
  fields ((declare a
           before-fetching (foo)
           after-replacing (bar newvalue))))

(structure s2 is-a s1
  fields ((declare a
           before-fetching (s2foo)
           after-fetching (s2bar newvalue))))

(replace an-s2 a 10) => (let
  ((datum an-s2)
   (newvalue 10)
   (%%ret (rplaca datum newvalue)))
   (bar newvalue)
   (s2bar newvalue)
   %%ret)
\end{verbatim}
8.1.4.3 must-be

The **must-be** attribute is modifiable using the standard syntax. The concatenation used is to queue the values and then, at the time the replace function is generated, 'and' the values together as shown below:

```
(structure s1
   fields ((declare a
             must-be (evenp newvalue)))

(structure s2 is-a s1
   fields ((declare a
             must-be (< 10 newvalue)))

(replace an-s2 a 20) ==> (rplaca
   an-s2
   (let ((newvalue 20))
     (cond
      ((and (evenp newvalue)
           (< 10 newvalue))
       newvalue)
      (t (must-be-handler
           'a
           newvalue
           '(s1 s2)
           an-s2)))))
```

8.2 Inheritance and typing

The effect Structure Inheritance has on typing pertains to what is returned by the **type** function. For a top level structure such as:

```
(structure level-1 ...)
```
3 July 1984

a call to **type** such as:

```
(type (create level-1))
```

returns a list whose only member is **level-1**. For the next level:

```
(structure level-2 is-a level-1 ...)
```

a call to **type** such as:

```
(type (create level-2))
```

will return a list of 2 elements, the first being the name of the **structure** that the form is an instance of, and the second being the parent of that structure.

In general the value returned by **type** is a list structures:

```
(struct1 struct2 ... structn)
```

such that **struct1** is the structure type of the form and for each **structi** in the tail of the list, **structi** is the parent of some **structj**, for

```
1 <= j < i
```
9 - Modes of Operation

Many of the features offered by CADRS have two or more different modes of operation. These modes are either explicitly set by the user or are set by CADRS based on the relationships between the different structures being defined. This chapter covers the different modes of operation, explains how they are explicitly set, and describes the conditions under which modes are changed by CADRS itself.

9.1 Macro Replacement

If the value of the symbol `cadrsReplaceFlag` is set to `t`, all calls to the user-level functions will replace themselves (using rplaca and rplacd) with their expansions. By default, the value of this symbol is `nil` and no replacement is done.

Of course, if efficiency is a major consideration, the best thing to do is to compile all functions and macros. However, since many system spend quite some time in interpretive mode before finally being compiled, this feature will result in a considerable amount of savings, especially when dealing with complicated structures.

This feature must be explicitly set by the user.

---

1. Remember that all top-level functions are written as macros.
9.2 Typing

Normally, CADRS will generate instance typing code only in the following situations:

1. When the value of the `typecheck` structure attribute is explicitly set to `t` in the structure definition.

2. When two or more structures contain conflicting definitions of the same field. In this case CADRS will force the value of `typecheck` to `t` for each structure.

Typechecking can be turned on for all structures by setting the value of the symbol `cadrsTyping` to `t`. The default for this symbol is `nil`.

9.3 Path Checking

CADRS has the ability to check the paths specified to fetch/replace type functions for correctness. To do this, both typing must be on (i.e. the symbol `cadrsTyping` must be set to `t`), and the value of the symbol `cadrsPathCheck` must be set to `t`. This feature must be turned on explicitly and is turned off (value of `nil`) by default.
10 - Exception Handling

The method for handling exceptions (errors) in CADRS is similar to the method used in Franz Opus 37 with an INTERLISP extension which makes the job of writing exception-handlers less difficult\(^1\). The exception handling algorithm is:

1. Check to see if there are any exception handlers for this class of error. If not go to step iii.

ii. Call all user specified exception handlers with the arguments for this class of error. If any return a non-nil list, return the first element of the list as the value to be used to continue processing.

iii. If none of the exception handlers 'handle' the error (or if none were specified), call the default error handler for this class of error.

10.1 CADRS exceptions

The following describes the different exceptions or recoverable errors implemented in version 1.00 of CADRS. Each exception is presented along with the arguments that will be passed to any exception handling function.

10.1.1 cadrs-except-must-be

This exception occurs when a must-be test for a field in a structure fails. All exception handlers for this error must except the arguments:

---

1. Exception handling is not tied deeply into the Franz error handling system to allow portability of CADRS to other LISP systems. The Franz error system is used only at the lowest level to facilitate debugging of errors.
3 July 1984

1. The name of the field the exception is generated for.

2. The value that caused the exception

3. The lineage of the structure (i.e. the value that would be returned by type).

4. The form that generated the error (i.e. the form that the replace/val-to was being performed on).

5. The s-expression that failed.

If the handler returns a non-nil list, the first value in the list is taken as the value to perform the replace with. Note that this value is not re-validated.\(^1\)

10.1.2 cadra-except-type?-nostruc

This exception is generated when one or more of the structures listed in the second argument to the type? function are not defined stuctures. All exception handlers for this exception must take 3 arguments, the first two being the arguments to type? and the third being the structure that was undefined.

If the handler returns a non-nil list, the first element is returned as the value of the call to type?.

\(^1\) This provides a means of overriding must-be checks
3 July 1984

10.1.3 cadrs-except-nfft

The 'no field for type' (nfft) exception occurs under the following conditions:

1. A fetch/val-of or replace/val-to is being performed on a structure instance.

2. Typing is turned on for this instance.

3. The structure type of the instance does not contain the field in question.

Exception handlers for this class of exception take 3 arguments; the first being the name of the field that caused the exception, the second the instance of the structure involved, and the third the function being performed (either val-of or val-to). If the handler returns a non-nil list, the first element will be returned.

10.2 Establishing an exception handler

Exception handlers are activated, or hooked in to CADRS by means of the establish function. The syntax is:

(establish exception-name handler)

where:

- exception-name is the name of the exception (as described above).

- handler is either a lambda or a symbol whose function binding is a lambda that will handle this type of exception.
3 July 1984

At this point, all establish does is to put the function name or lambda in the list that is kept as the value of the symbol exception-name. establish should be used to maintain the upward compatibility of the system with future versions of CADRS.

10.3 Sample Exception Handlers

The following demonstrates how one would write exception handlers for a few of the different types of exceptions.

10.3.1 cadrs-except-must-be Example
3 July 1984

The application is to build a structure that will prompt for either "yes" or "no" and return t or nil respectively.

(structure yes-no
  fields ((declare answer
to-replace (memq newvalue
'(Y y Yes yes YES))
must-be (memq newvalue
'(Y y Yes yes YES
N n No no NO)))))

(defun yes-no-handler (field value type form test)
  (if (memq 'yes-no type)
    then (do ((newvalue (prog2
        (msg N
"Invalid answer"
N
"Answer yes or no : ")
(read)))
 (eval test) (ncons newvalue))
    else nil))

(establish cadrsexcept-must-be 'yes-no-handler)

Note the following:

1. If the exception handler is designed to handle exceptions for a specific structure, field, or structure/field combination, it must check the value of the first and third (field and type) arguments to determine that the exception was generated for that type. If this is not the case, the handler should return nil as shown.

2. The handler must validate the value that it returns as CADRS does not.

3. The handler must return values as the first value in a list (done in this example with (ncons newvalue).
3 July 1984

10.3.2 cadrs-except-type?-nostruc example

Here the example in the chapter on type and type? is revisited using an exception handler.

(structure person
  fields ((declare name is-a name)
          age
          sex))

(defun type?-women (form list structure)
  (if (and
       (eq structure 'women)
       (memq 'person (type form)))
     then (ncons (eq (fetch form sex) 'female))
     else nil))

(establish cadrs-except-type?-nostruc 'type?-women)

(type? (create person (sex . 'female)) (woman)) => t

Note again that type?-women checks that the exception it is handling is for a women structure and returns the value as the first element in a list using ncons.

1. This time women is not defined as a structure
3 July 1984

11 - CADRS Utilities

This chapter describes the utility packages provided for version 1.00. Included are:

1. A validation script for version 1.00 for Franz Lisp.

2. A reset function for CADRS.

3. A syntax macro that provides an abbreviated form of fetch and replace.

4. Other useful functions.

11.1 CADRSVAL

The CADRSVAL package provides a series of functions which serve to test each of the features of CADRS. [[not yet complete]]

11.2

The cadrs-reset function (currently in CADRSRTS.L) will remove the definitions for all currently defined structures. The call to the function is:

(cadrs-reset)

11.3 RPM

The RPM package written by Tim Finin of U. of P. provides an abbreviated form for writing fetch an replace statements. Once RPM is loaded in with CADRS, the following constructs are accepted:
3 July 1984

1. \#struc.path
   This expands into (fetch struc path)

2. \#struc.path=newval
   This expands into (replace struc path newval)

11.4 CADRSUTIL

   CADRSUTIL provides a set of functions to help with debugging and testing of
   CADRS structures and functions that use them. Each function is described below:

11.4.1 structures

   structures is a function of no arguments that will display a list of the
   currently defined structures.

11.4.2 dump-field

   dump-field will display the current attribute values for a field as it relates
   to a given structure. The format is:

   (dump-field field struc)

   where:

   o field - is the name of the field to be dumped.

   o struc - is a structure containing that field.
3 July 1984

11.4.3 dump-structure

    dump-structure - displays all attributes for a given structure. The format is:

    (dump-structure struc)

where:

  o struc - is the structure to be dumped.

11.4.4 dump-all-structures

    This is a function of no arguments that dumps all attribute values for all
    currently defined structures.

11.4.5 edits

    The edits function uses the LISP editing functions to alter a structure
    definition in the LISP environment. The format is:

    (edits struc)

where:

  o struc - is the structure to edit.
12 - Conclusions

12.1 Concluding Remarks

Through the extension of the traditional structure definition formalism, CADRS embodies many of the strengths of an object oriented system in an efficient implementation. The package has a significant number of features not found in other LISP-based structure definition packages, including features imported from network representation schemes.

12.2 Future Work

There are a number of additions to CADRS under consideration; these being:

i. a more flexible (English-like) structure definition language to replace the rigorous attribute/value syntax.

ii. to build, in conjunction with (i), a more robust structure access mechanism to handle data-path inference, run-time specifications of data-paths, and the ability to specify a structure type as part of the path to clear up ambiguities.

iii. to possibly build a typed function formalism (a la GLISP), in conjunction with (i) and (ii).

iv. to build a message facility to allow more complete object centered programming.

1. These ambiguities currently arise when 2 structures contain 'different' definitions of the same field'. This situation is solved by typechecking the instance when performing the field access.
3 July 1984

v. to add a destroy function to destroy structure instances (necessary if CDRS is to be used in database research).

vi. a means to redefine (in a COBOL sense) parts of a structure.
References

1. Bawden, Alan. "DEFSTRUCT Reference Documentation." Massachusetts Institute of Technology (ALAN@MC), 1980


