MIT/LCS/TM-15

AN EXPANSION OF THE DATA STRUCTURING

CAPABILITIES OF PAL

Stephen N. Zilles

October 1970

AN EXPANSION OF THE DATA STRUCTURING CAPABILITIES OF PAL

Stephen N. Zilles

MAC Technical Memorandum 15

October 1970

(This report was reproduced from an S.M. Thesis, MIT, Department of Electrical Engineering, June 1970)

Massachusetts Institute of Technology

PROJECT MAC

545 Main Street Cambridge 02139

ACKNOWLEDGEMENT

The work described in this thesis was in part supported by Project MAC and I am grateful for their financial assistance.

I wish to thank my thesis advisor, Professor John Wozencraft for his help in identifying important problems, his criticism of weak arguments, his helpful suggestions, and his encouragement. I also wish to thank Professor Arthur Evans for helpful comments and suggestions. Discussions with Bob Thomas and D. Austin Henderson helped clarify some of the ideas presented herein and I greatfully acknowledge their intelligent criticism.

My thanks also go to Miss Betsy Rutter who had the patience to struggle through a mound of cryptic handwriting and produce this typed manuscript.

Finally, I wish to thank my wife, Connie, not only for her editorial assistance but also for the encouragement, faith and understanding which made it possible to complete this work.

Stephen N. Zilles
Waltham, Mass.
June, 1970

CONTENTS

ACKNOWLEDGEMENT		3
I.	INTRODUCTION	6
	A Design Principle	7
	Background	8
	Overview	9
II.	AN ALTERNATIVE TO AUTOMATICALLY DEFAULTING TO LVALUES	12
	The Implications of the Change to Location Generation in PAL	22
	Alternatives for Passing Arguments	28
	Modifications to the L-PAL Gedanken Evaluator	31
	Other Alternatives for Handling Assignment	33
III.	REPRESENTING DATA STRUCTURES BY FUNCTIONS OVER SYMBOLIC DOMAINS	41
	Representing Data without a Natural Ordering	41
	The Properties a Data Structuring Facility Should Possess	44
	Landin's Structure Definition	46
	The S-PAL Representation of Data Structures	50
	Atoms	54
	The Special Properties of Data Functions	58
	Universal Constructors	64
IV.	THE FORMAL DEFINITION OF DATA FUNCTIONS	67
	Simple Structure Definitions	67
	Predicates for Types with Alternative Forms	71
	A Representation for the Primitives MakeStr and IsStr	77
	A Syntactic Abbreviation which Defines a Constructor and Complex Predicate	84
	Data Functions with a Mixed Domain	88
	Structures with Explicitly Enclosed Substructures	94

	The Standardization Process	110
	An Alternative to the Ordered Tuple as the Constructor Argument	122
V.	A TYPE SYSTEM FOR STRUCTURES	130
	Dynamic Versus Static Type Systems	132
	A Type System Based on Predicates	135
	The Syntax for Verifiers	141
	Using Verifiers in The Constructor	148
	The Problems Associated with Unrestricted Verifiers	151
	The S-PAL Solution	160
VI.	CONCLUSIONS AND ANALYSIS	162
	Treating Locations as Values	162
	Functional Data Structures	164
	A Type System Based on Predicate Functions	165
	A Possible Implementation for Data Functions	167
	Possible Modifications to S-PAL and Future Directions	170
Appendix A.	The Complete S-PAL Syntax	182
Appendix B.	The Gedanken Evaluator for S-PAL	184
Appendix C.	The Standardizing Functions for Typeless S-PAL	193
Appendix D.	The Representation of S-pal Data Functions	195
REFERENCES		198

Chapter I

Introduction

PAL is a language designed for use as a tool to help teach programming linguistics [8]. As such, it incorporates generalizations of many of the features that are found in most common programming languages. PAL also has a relatively compact formal semantic definition. However, careful reading of this definition clearly shows that it would be much more readable if the control items and abstract syntax could be represented with a more sophisticated data definition facility. One goal of this thesis is to present such a facility.

But, the objective is not just to present the formal definition in a readable format. More importantly, we are interested in investigating the suitability of the PAL formal definition techniques for describing data structures. We will show that it is possible to integrate a facility for data structures into the L-PAL subset. The formalization of this facility is analogous to the formalization of the existing PAL definitional facilities.

Another objective of this thesis is to increase the flexibility of PAL and to give the user more control over the form and use of his data. The features we will add make stronger representations of the data structure possible. In particular, the introduction of type checking and tags in all data structures makes it possible for the user to limit the properties of the data structures and to enforce these limitations. Finally, changes to the handling of locations increase the users control over their creation.

A Design Principle

The research presented in this thesis is only an initial step toward a satisfactory facility for structuring data. There are many problems, some of which are discussed in the thesis, which we leave unsolved. The whole area of data structures is a bottomless pit where each foray raises as many problems as it solves. Because there are so many paths to explore it is necessary to adopt criteria for deciding when to terminate an exploration.

The criteria we have adopted are simplicity and generality. We have attempted to stop when there is no obvious continuation to the work and when the facilities we have proposed allow the user to implement his own specialization. It seems both futile and impractical to provide special solutions for every possible viewpoint. Therefore, when there is no one solution which is clearly preferable to all other solutions we have tried to move back one step and to adopt a simple approach which is general enough to implement the proposed solutions.

Unfortunately, we have not always succeeded in applying these criteria. We have proposed some additions that appear to be excessively complicated for the additional facilities they provide. The area of types is perhaps where these criterea have been most successfully applied. However, we have also avoided introducing many of the specializations suggested by other authors when they could be implemented within the existing language framework.

Background

Most existing programming languages include some facilities for building data structure. However, there is no uniform agreement on a suitable set of functions to include. Standish[33] has surveyed most of the work prior to 1967. Hence, we will only update that survey to the present. The relevant background material can be devided into three categories.

1) The majority of the work has been in defining suitable notations for describing the data structures. Most of this work (Earley[7], Hoare[11], Standish[33]) has been general purpose and language independent, but some more formal descriptions (Laski[18], Lucas[20]) of a particular case occur. It is also necessary to mention the existance of several general purpose languages (POP-2[4], BASEL[10,12], ALGOL68[37], AMBIT/G[5]) which have included powerful facilities for data structuring.

2) A given description usually has many possible representations. Several authors have discussed the problem of representation in both machine dependent (Earley[7], Laurence[19], Vigor[38]) and abstract

terms (Balzer[1], Park[26], Reynolds[30]).

3) A limited amount of work in the formalization of the semantics of data structures has occurred. Park[26] explored the formal properties of assignment in data structure. The majority of the other work has been incidental to the formalization of the languages (BASEL[10,12], GEDANKEN[30], ALGOL68[37]) in which the data structure facilities are embedded.

In addition to this general background material, several authors had a particularly strong influence on the form of the S-PAL extensions. The syntax and content of the structure definitions is drawn from the work by Landin[13,16,17] in describing data structures. The representation is a generalization of the functional data structures of Reynolds[30]. The approach was also influenced by the structural facilities of COBOL[36] and PL/I[27]. The type system is largely novel, but the tags used in S-PAL also occur in the work by Standish[33], and in similar forms in Reynolds[31] and Morris[25].

The formal definition and the extensions themselves are based on PAL. Because we must build on previous work we will assume that the reader is familiar with the PAL language and its method of formal definition. In particular, chapters 2 and 3 of reference [40] should be sufficient background.

Overview

The extended language is called S-PAL for Structural PAL.

The extensions are presented both informally with examples and through modifications to the formal definition of PAL. Most of the formal definition of S-PAL is encoded in terms of the R-PAL subset (i.e., assignment is not used). This was done because it did not appear to complicate the definitions and served as a demonstration that the data structures required only the R-PAL subset for their definition. Hence, these additions could be combined with the L-PAL additions to create an expanded L-PAL with data structures.

Chapter II of this thesis begins our development with a description of an extension to the handling of locations. The current PAL approach is reviewed and an alternative approach which treats locations as another type of value is presented. The consequences of this change and some alternative formulations are discussed.

The facilities for structuring data are described in Chapters III and IV. In Chapter III the concept of a data function is introduced and some of its more important attributes are described. The requirements of a suitable representation for data structures are presented. Some alternative representations are discussed and it is shown that the data functions meet these requirements. The formal definition of structure definitions and how they are transformed into data functions is given in Chapter IV. The full capability of structure definitions is developed in several steps, in which each step adds facilities to those presented in the preceding step. The chapter ends with a generalization of the argument list of a function.

A novel approach to type checking systems is discussed in Chapter V. The reasons for restricting the discussion to dynamic type checking are presented and a type system based on predicate functions is formally defined. Some of the consequences of this approach are discussed.

The important ideas and conclusions of the preceding chapters are summarized in Chapter VI. An approach to implementing data functions and possible extensions of this work are also presented and discussed.

Chapter II

An Alternative to Automatically Defaulting to Lvalues

Introduction

This Chapter presents an alternative method of handling memory locations in PAL. The current PAL definition distinguishes memory locations from the abstract objects (obs) which may be contained in the memory locations. The memory locations are called Lvalues and the objects are called Rvalues because they are the values required by the left and right sides of an assignment statement.

It is obvious that an Lvalue is more general than an Rvalue since the Rvalue may always be obtained if the Lvalue is known. However, it is not in general possible to find the Lvalue in which a particular Rvalue is contained. PAL currently holds to a design decision which forces Lvalues wherever they are reasonable to preserve the greatest generality.

The effect of this design decision has been to establish contexts in which Lvalues or Rvalues are required. It is unreasonable to always require an Lvalue context since it may be of no utility or even an inconvenience. For example, in evaluating the expression X+3, only the Rvalues of X and 3 are needed to compute their sum. Also it is not always reasonable to yield an Lvalue as a result. If the above sum occurred in another sum, say y+(x+3), then there is no need to produce an Lvalue for x+3. Hence the natural result of a basic function such as addition is an Rvalue.

It is to a certain extent a value judgement as to where the generality of Lvalues should occur. The principle of consistency is used to give an Lvalue context to anything which might naturally occur on the left hand side of an assignment statement. This includes both identifiers and the components of a tuple. In this way almost everything is updatable.

While the context of an expression determines what mode, Rvalue or Lvalue, is required, the form of an expression determines which mode actually results from the evaluation. When the contextual mode differs from the resulting mode a transfer function is automatically inserted to give the correct contextual mode. The mode contexts are given in Table II.1 while Table II.2 gives the modes resulting from the expressions.

Since Lvalues are used when variables are bound it is possible for two variables to designate the same Lvalue. This is called sharing. To make it possible to avoid sharing, the operator "$" is used to extract the Rvalue from its single argument. When $ is applied to an Rvalue the result is that Rvalue. But when $ is applied to an Lvalue the Rvalue contained in that Lvalue is the result. Note that when $ occurs in an

Table 1: The current mode context table

R α R β R $ R   L % <variable> L

R aug L   L $\{ , L \}_1^\infty$

R L   R -> B | B

test R ifso B ifnot B

test R ifnot B ifso B

if R do L   while R do L

goto R   R ; B   L := R

let <definition> in L

L where <definition>

valof L   res L

(B)   [B]

fn <bv part> . L

<variable> $\{ ,$ <variable> $\}_0^\infty$ = L

<variable> <bv part> = L

Table 2: Current table of resulting modes

R-type expressions

<quotation>   <numeric>   <literal>

$ E   E α E β E

E $\{ , E \}_1^\infty$ E aug E

fn <bv part> . E

E := E

L-type expressions

E E   <variable>

E % <variable> E   valof E

The symbol R indicates that an Rvalue context occurs and similarly L indicates an Lvalue context. B indicates that there is no automatic conversion of values performed.

Lvalue context a new Lvalue is created to hold the resulting Rvalue so unsharing is accomplished.

An Alternative to Automatic Handling of L and Rvalues

The main thesis of this chapter is that it is not necessary always to force Lvalues to be created in certain contexts. In fact, it is possible to leave the decision on Lvalue creation strictly to the user. This latter approach has several advantages.

1) If Lvalues are not always forced then it would be possible for identifiers to be bound to Rvalues. This has the advantages that less storage space may be needed and that the value of the variable will remain constant. Hence it will be possible for the compiler to optimize references to that variable.

2) Since the value of variables bound to Rvalues is fixed, it provides data integrity. The variable cannot be updated by assignment because no location is associated with the variable.

3) Allowing Rvalues as well as Lvalues as parameters to functions gives greater control over the possible effects of the function. It is not

possible to update an Rvalue parameter.

With the above advantages as motivation it appears that the natural way to put locations under user control is to add the locations to the set of basic obs. To do this it is necessary to axiomiatize the desired properties of locations.

Ax II.1 There exists a countable set of locations which are distinct from all other obs.

These locations are distinct from each other and by the countable property it is possible to assign to each location an integer which identifies that location. In normal terminology this integer is called an address.

The main use of a location is to hold a value. Therefore, the remaining axioms are primarily concerned with the relationship of locations to other obs. The term <u>memory</u> is introduced to represent the relation "location α holds Rvalue β". Because the computations we are interested in are of necessity finite processes, memories are defined only on finite subsets of the set of all locations.

<u>Definition</u> A memory is a finite set of (location, Rvalue) pairs with the property that each first component is distinct from every other first component in the memory.

The memory can be viewed as a finite function from a subset of the set of locations into the set of Rvalues. Since every two pairs have distinct first components, a location may "hold" only one Rvalue in any particular memory. Hence, the function is well defined over the set of locations in the memory.

Ax II.2 There is a function **Contents** such that if μ is a memory and α is a location in the memory then

$$\text{Contents}\ (\mu, \alpha) = \mu\alpha$$

and is otherwise undefined.

This function is used to obtain the Rvalue currently held in location α. It is undefined on locations not in the memory for practical reasons. As noted above, memories are finite because the computations of interest are finite. This restriction to finiteness is analogous to the use of a Turing machine storage tape. At any particular step in the computation only a finite number of squares have actually been scanned. Hence, even though the computation is unbounded and may eventually use an infinite amount of tape, at any instant it depends only on a finite amount of tape. Therefore, it cannot distinguish whether the tape was initially infinite or if instead a new tape square is appended whenever the Turing machine is about to use the last square of the current tape. This latter approach more closely models a physical machine and justifies the restriction to finite memories.

The contents of the tape squares which have not been scanned are unimportant. They only become important when they are about to be scanned. Hence, it is only necessary to initialize them when they are appended to the tape. This justifies the decision to define the **Contents** function only on the locations in the memory. The question of initialization of memory location is delayed until the axiom for memory extension are presented.

Since a memory may associate only one Rvalue with a location it is necessary to provide a function which will produce memories with the locations holding different Rvalues. This function complements the contents function. Referential transparency is preserved by creating a new memory instead of modifying the old one.

Ax II.3 There exists a function <u>Update</u> such that if

μ is a memory

α^* is a location

ω is an Rvalue(i.e., not a location)

Then ν=Update(μ,α^*,ω) is a memory such that

$$\text{Contents}(\nu,\alpha) = \begin{cases} \text{Contents}(\mu,\alpha) & \text{if } \alpha \neq \alpha^* \\ \omega & \text{if } \alpha = \alpha^* \end{cases}$$

This function produces a new memory in which the location

α* holds a new Rvalue ω. It is important to note that a location is intentionally prohibited from holding another location as its Rvalue. Or in other words, a location is never an Rvalue. This is certainly not the only possible way to treat locations. Many current languages which have the concept of locations allow locations to hold other locations. For example, this is the case in ALGOL 68 [37], BASEL [10,12], and GEDANKEN [30]. The main reason for not allowing locations as Rvalues is motivational. The memory location is a place which holds a value. It is analogous to the piece of paper on which a value can be written. Since it does not appear to make much sense to talk about a piece of paper which holds another piece of paper, the analogy leads to restricting locations from holding other locations. The implications and alternatives to this choice will be discussed in greater detail later in the Chapter.

The locations are metalinguistically distinct by definition. However, it is possible to bind different names to a single location, so the user must be able to test when two names are bound to the same location. For this purpose we will say two locations α and β are distinct if and only if μ is a memory and ω_1 and ω_2 are Rvalues such that

i) $\text{Contents}(\mu,\alpha)\neq\omega_1$ and $\text{Contents}(\mu,\beta)\neq\omega_2$

ii) $\text{Contents}(\text{Update}(\mu,\beta,\omega_1),\alpha)=\text{Contents}(\mu,\alpha)$

and $\text{Contents}(\text{Update}(\mu,\alpha,\omega_2),\beta)=\text{Contents}(\mu,\beta)$

Hence, two locations are distinct when updating one location does not affect the contents of the other locations.

Ax II.4 There exists a memory $\mu_{\#}$ with an empty domain.

Ax II.5 There exists a function Extend such that if μ is a memory

$$\text{Extend}(\mu)=(\nu,\alpha^*)$$

where

(i) $\text{domain}(\nu)=\text{domain}(\mu)\cup\{\alpha^*\}$

(ii) α^* is distinct from every location in the domain(μ)

(iii) $$\text{Contents}(\nu,\alpha)=\begin{cases}\text{Contents}(\mu,\alpha) & \text{if } \alpha\neq\alpha^* \\ \# & \text{if } \alpha=\alpha^*\end{cases}$$

These axioms introduce the concept of a memory extension. The memory begins as an empty function and through the use of Extend the memory function is augmented with new locations distinct from all the other locations already in the memory. Each new location is initialized to hold a special value designated by #. The Extend function returns two values (a 2- tuple) since both the new location and the new memory are needed.

Actually, the above axioms are almost the same as the axioms for memories in the current PAL definition. The main change was the introduction of a specific prohibition against locations holding locations. The important changes to PAL are made in the context rules which determined when Lvalues will be created. Giving locations the status of obs means there is no longer a need to restrict the binding of identifiers solely to locations. In consequence, the results of expressions which do not produce Lvalue results will not be automatically converted to Lvalues. Since it is unreasonable to do without Lvalues altogether, a new operater <u>loc</u> is introduced to allow explicit creation of Lvalues. The operater <u>loc</u> obtains a new location using Extend and puts the Rvalue which is its argument into the new location. The result is the updated location. Since the argument of <u>loc</u> must be an Rvalue, an Rvalue context is forced. Therefore, an expression such as <u>loc</u>(<u>loc</u> 3) creates two new locations each of which holds a 3 since an automatic application of Contents is used to obtain the Rvalue 3 after the first application of <u>loc</u> The location resulting from the first application of <u>loc</u> becomes inaccessable because the Contents function does not pass on the Lvalue of its argument. Thus, <u>loc</u> performs the same function as $ performs in an Lvalue context in the current PAL.

While Lvalues are not automatically created it is still necessary and reasonable to insert automatic transfers from Lvalues to Rvalues. For example, the right hand side of an assignment statement and the argument of loc both require an Rvalue. The relaxation of Lvalue contexts has produced. more contexts which force neither L or Rvalues. Therefore, the operator val is introduced to extract explicit Rvalues. The argument of val may be either an Lvalue or an Rvalue. If it is an Rvalue the result of val is that Rvalue. If the argument is an Lvalue, the result is the Rvalue which is the contents of that Lvalue. The modified context and form rules are given in tables II.3 and II.4

The Implications of The Change to Location Generation in PAL

One of the primary functions of locations beyond that of allowing assignments is to allow several identifiers to share the same location. Sharing means that an update to one identifier changes the Rvalue associated with the identifiers that share with it. In the current PAL, sharing occurs naturally and the $ operator must be used to prevent sharing

Table 3: The new mode context table

R α R   β R   val R   B % <variable> B

R aug B   B { , B $\}_1^\infty$

R B   R -> B | B

test R ifso B ifnot B

test R ifnot B ifso B

if R do B   while R do B

goto R   R ; B   L := R

let definition in B

B where definition

valof B   res B

(B)   [B]

fn <bv part> . B

<variable> {, <variable>$\}_0^\infty$ = B

<variable> <bv part> = B

Table 4: New table of resulting modes

R-type expressions

<quotation>   <numeric>   <literal>

val E   E α E   β E

E { , E $\}_1^\infty$ E aug E

fn <bv part> . E

E := E

L-type expressions

loc E

B-type expressions

E E   <variable>

E % <variable> E   valof E

The symbol R indicates that an Rvalue context occurs and similarly L indicates an Lvalue context and B indicates that there is no automatic conversion of values.

from occurring. Because locations must be explicitly created in S-PAL, sharing occurs only when a location is bound to the identifiers.

The above statement is somewhat deceptive since sharing is defined solely by the effect of an update operation. The real reason that sharing does not occur unless identifiers are bound to Lvalues is that updates are not possible to variables which are bound to Rvalues. The update function is only defined in Lvalues. Hence, it is reasonable to introduce the term constant (or manifest constant[29]) for identifiers which are bound to Rvalues and to reserve the term variables for identifiers bound to Lvalues. Because there are no Lvalue contexts, it is necessary to define what happens when an Rvalue occurs on the left hand side of an assignment. This problem does not arise in the current PAL because the Lvalue context always assures an Lvalue will occur on the left hand side of an assignment. This means that the assignment 3:=5 will have no effect because a new location is created to hold 3 and the assignment changes its contents to 5. However, the location is inaccessable following the assignment so no noticable effect occurs. There are essentially two choices on what to do with Rvalues on the left of assignments. One action is to simulate the effect of creating a new location, assigning to it and forgetting the location. This form of assignment is nugatory on all constants and constant identifiers. The other alternative is to raise an error condition whenever an Rvalue is on the left of an assignment. I feel the latter action is better since with the generality of PAL it is very simple to make horrible mistakes and any action which helps to find these mistakes sooner is very useful.

Removing the automatic creation of Lvalues from PAL also has an effect on the construction and augmentation of tuples. Previously the range of a tuple was restricted solely to Lvalues. This meant every component of a tuple could share and was updatable. In S-PAL the range of a tuple is extended to be any ob in the universe of discourse. This means that it is possible to create tuples whose components are all Rvalues or even mixed Rvalues and Lvalues. Therefore, certain components of a tuple may not be updatable.

The <u>aug</u> operation does not modify previously constructed tuples since this would destroy referential transparancy. Instead, <u>aug</u> produces a new tuple of length n+1 whose first n components are the "same" as those of the previous tuple and the n+1st component is the augmented component. To be complete it is necessary to specify what is meant by "whose first n components are the same as those of the previous tuple's".

This is simply solved in the current PAL by requiring that all components of a tuple be Lvalues. Then the first n components of the new tuple share with the corresponding components of the old tuple. Hence, the components designate the "same" values.

The same solution works for Lvalued components in S-PAL. It is the Rvalued components which raise problems. An Lvalue or more explicitly a location is a very simple data object. Two locations are equal if and only if they share. However, Rvalues are both simple, such as reals or integers, and complex such as tuples or functions. While equality is defined naturally for simple Rvalues, the PAL programmer must define what he means by equality for the complex Rvalues. There is no built in definition of equality for functions or tuples.

One alternative for handling Rvalues in tuples is to copy the Rvalue and use the copy in constructing the new tuple. We choose to define a copy to be the "same" as the original if and only if it produces the same result as the original under every operation which is applicable to the original. In particular, this definition requires that assignment to any subpart of an Rvalue must affect the copy and the original in the same way. This means that the copy is made by copying the structure only as far as locations or simple Rvalues. This is natural since equality is defined for these simple values, so if the structure connecting the values is identical, the copy and the original must be the "same".

If the structure is identical up to the locations it cannot be modified by any subsequent operations. Updates can only affect the contents of a location, and the copy and original Rvalue share the same locations. Therefore, it is unnecessary to copy the Rvalues in the first place. This facilitates implementing S-PAL since much less than a full copy is needed to perform the aug operation. The new tuple is constructed by copying only the map between the n integers of the original and their associated values and extending it to include the new value as the n+lth component. Thus, a one level copy suffices to duplicate the original tuple.

Because the tuple is copied before being augmented, it is impossible to modify a tuple occuring as an Rvalue in another tuple. This is consistent with the treatment of other constants. It also means that it is impossible to put loops into data structures without using an assignment operation. This is because no previously defined object can refer to the newly constructed tuple unless the new tuple is assigned to that object.

Alternatives for Passing Arguments

Distinguishing between variable and constant bindings makes possible a number of different ways of passing arguments and handling formal parameters. Whether an argument will be modified or not can be controlled by either the calling or the called function. When constant arguments are used, the called function can not produce side effects by assigning to the formal parameters. Within the called function, the formal parameters may be either bound to the argument or to a location which holds the argument. In the former case assignments to the parameter are impossible since it is bound to an Rvalue and updates are not allowed. In the latter case, the formal parameter is more like a local variable which is initialized to the Rvalue of the argument. In this case assignments only change the local value and have no affect on the argument.

If the passed argument is an Lvalue more alternatives are possible. If the formal parameter is bound to a new location containing the argument as in the second case above, the called function cannot distinguish between Lvalues and Rvalues arguments. In either case the affects of the formal parameter are local to the function. This corresponds to the ALGOL form of "call by value."

If updates to formal parameters are to be forbidden as in the first case above, the formal parameter may be bound to <u>val</u> of the passed argument. Then no matter whether an L or Rvalue was passed the binding is always to the Rvalue. This way guarantees that the arguments to a function will remain constant for the duration of the function invocation.

The final alternative is to bind the formal parameter to the argument just as it was passed. Then if an Lvalue was passed, side effects through updates are possible. This corresponds to what is called "call by reference" by Strachey [35] There is a slight difference, however, because the caller has control of whether an Lvalue is passed. Therefore call by reference becomes a cooperative effort between the calling and the called function.

The handling of free variables is another aspect of functions that is discussed by Strachey. The value of a function definition is a λclosure. The λclosure contains all the information necessary to evaluate the function. This consists of the text of the function and the values to associate with any free variables in the function. There are a variety of ways of handling free variables, two of which are used in CPL[2]. The values for the free variables in the λ-closure form the free variable list. In CPL and other languages a free variable list is built when the λ-closure is made and the identifiers associated with the free variables are bound to the values on the free variable list. This is, they are bound to offset in the free variable list. If the function is defined with the operater "≡" the Lvalues of the associated values are put into the free variable list. Alternatively if the operator "=" is used the Rvalues of the free variables are used to build the free variable list.

In PAL the identifiers are not bound to the values in the free variable list, but instead the free variable list consists of all the free identifiers and their bindings when the function was defined. When a PAL function is invoked the values of the free variables are obtained by searching for the identifier in the free variable list and using the value that identifier is bound to. Since the current PAL only allows Lvalues in bindings, all definitions have the same affect as "≡" definitions in CPL. However, in S-PAL Rvalues may also occur, so definitions fall somewhere in between the "≡" and "=" definitions of CPL.

It is difficult to create "=" type definitions in S-PAL. Even using <u>val</u> will not help because the argument of <u>val</u> is not evaluated until the function is invoked and the current value of the argument will be used. The only way to achieve the affect of Rvalues on the free variable list in S-PAL is to define the function in an environment where all the free variables are already bound to Rvalues.

Modifications to the L-PAL Gedanken Evaluator

Relatively few modifications are necessary to make Lvalues objects in L-PAL. The main change is to remove the Lvalue contexts as has been already noted. The Lvalue contexts are forced in only two places in the gendanken machine, namely, in the Extendtuple function and the Apply λclosure function. These are the only places where any form of binding occurs in the gendanken evaluator. These functions are simplfied by removing the test for Lvalues and the associated invocation of NewLval to build an Lvalue if none was present. See appendix B for the modification.

The above modifications remove all uses of NewLval but it is used in the new definition for loc. Similarly a definition for val replaces the $ operator. The two new steps in Transform are

```
| x eq 'loc'          → NewLval(A)
| x eq 'val'          → Stepcontrol(A)
```

replacing

```
| x eq '$'            → Stepcontrol(A)
```

Note that since the action for loc occurs below the R context forcing, NewLval will always be acting on an Rvalue.

The final change is not as clean as the preceding changes. In the current PAL all basic functions have an Rvalue context. This is reflected in Applybasic which automatically extracts the Rvalue before applying the basic function. This is not possible in S-PAL since there are basic functions such as Isloc which require that automatic applications of val be inhibited. There are two possible solutions to the problem. The first solution is to allow basic functions to take both Lvalues and Rvalues as arguments. This would make basic functions more like user defined functions which no longer have context rules. However, this solution seems to introduce a certain amount of inefficiency in any implementation since every basic function using Rvalues would first have to check its arguments. If they were Lvalues it would have to extract the contents. This suggests an alternative solution which distinguishes two classes of basic functions. The first class of functions always takes Rvalue arguments so transfers are automatically performed. The second class of basic function tests its arguments so transfer functions are not needed and should not be inserted. This solution allows a compiler for PAL to insert transfer functions wherever they are allowed and needed. It can be affected by modifying the Applybasic function to be:

<u>def</u> Applybasic (C,S,E,D,M)=

<u>let</u> x = IsRfcn(t S) Rval(M,2nd S)

| 2nd S

<u>in</u> r C,Push[apply(t S)x,r2 S, E,D,M

The main disadvantage to the second solution is that the function Applybasic is relatively more complex. It now must test which type of basic function is to be applied. Of course, it is the possibility of making this test which allows the automatic insertion of a transfer function.

<u>Other Alternatives for Handling Assignment</u>

The literature is filled with a number of different proposals for formally defining the affect of assignment [3,4,12,26,34,39]. Some of the proposals are based on locations, while others either ignore the concept or modify it so it is unrecognizable. This section explores a subset of possible alternatives to S-PAL and discusses the differences.

<u>Syntactic Conveniences</u>

In S-PAL a location is never created without the explicit use of the <u>loc</u> operator. On the other hand in the current PAL a location is automatically created by defining a name. For example, the phrase

<u>let</u> X=2 in M

creates a new location, puts the value 2 into it, and binds it to the name X. In S-PAL this phrase would bind the name X directly to the value 2. If the user desires a variable which can be updated he must insert a loc operator as in

let X=loc 2 in N

Thus, it is syntactically easier to define "variables" in the current PAL than it is in S-PAL.

This distinction is more clearly seen in the equivalent lambda expressions. The first phrase is equivalent to $(\lambda X.M)2$ while the second is $(\lambda X.N)(loc2)$. Currently in PAL the argument of a λ expression is forced to an Lvalue so the desired location is created. But without forcing an Lvalue the binding of X will be to the constant 2.

There is an alternative solution to the problem which is found in CPL. Instead of associating an Lvalue context with the argument of a λ expression the right hand side of an "=" sign is desugared with the loc. That is "let X=2 in M" becomes $(\lambda X.M)(loc2)$.

If this were the only form for defining a binding then sharing and constants could not be obtained. Therefore, it is necessary to introduce a second definitional operator, such as the "≃" used in CPL, which does not force the creation of a location but just binds the name to the value (R or L) on the right hand side of the definition.

The above alternative was not chosen primarily for pedagogical reasons. There is a great value in making location creation explicit. Since they alone have side effects, pointing out their occurrences makes it easier to debug the programs and restricts unnecessary uses of locations. Also, having only one form of definition reduces the complexity of the language.

Should Locations be Able to Hold Other Locations?

In many languages where locations exist in the language it is possible for locations to be the values of other locations. This is specifically prohibited in PAL in part for reasons given earlier in the Chapter. However, it is useful to explore the other alternatives.

The reason given most often for allowing locations to hold locations is that of generality. The language designer can find no reason why locations _must_ be excluded from the set of Rvalues so they are allowed in the name of generality. However, generality is a vague concept in many applications. Often generality means allowing an object to appear anywhere it makes sense. Obviously all contexts do not make sense. For example, the sum of two strings of letters does not usually make sense. However, if the letters are assigned numeric values then, the sum might occur in some coding scheme. The problem is that what makes sense is a value judgement on the part of the language designer. It is my belief that locations holding other locations does not make sense. The main reason for this was given earlier in the Chapter using the analogy between a storage location and a piece of paper.

This analogy can be extended somewhat further to show a reasonable alternative to locations within locations. While a piece of paper cannot really hold another piece of paper it can hold a reference to another piece of paper. For example, a manuscript may hold the statement "for further discussion see page 257". This is a reference to another page and is a proper value for a page to hold. Hence by analogy a location should be allowed to hold a reference to another location. This is in fact possible in S-PAL or even the current PAL for that matter. In the PAL definition only the locations themselves are available to the user not their names. This allows greater freedom in choosing a particular implementation of the memory. If a programmer wishes to refer to a location he must give it a name. He can do this by binding the location to an identifier, but identifiers can not be the values of locations.

The other way a location is made accessable is by being a component of a **tuple**. It is possible to view the tuple as a generalization of the idea of pointers as found in PL/I[27].

While a pointer can only identify a single memory location the tuple can designate many distinct memory locations. Each component of the tuple can be a different location. The pointer corresponds to a 1-tuple. References to other locations can be implemented by assigning to the location a 1-tuple whose only component is the location being referenced. Thus, the tuple is also a means for "naming" locations.

It may appear that it is awkward to evaluate a tuple to be able to use the referenced location. However, this is really a problem inherent with references. Consider the following small excerpt of code for a language which allows locations to hold locations.

<u>let</u> X = <u>loc</u> 2 <u>in</u>

X:= <u>loc</u> 3;

X:= 5

When the block is entered, X is bound to a location holding the value 2. The first assignment changes the value held by the location X to another location which holds the value 3. Now does the second assignment modify the contents of the location X or does it modify the contents of the location refered to by location X? Because assignment requires an Lvalue and X is bound to an Lvalue it is natural to do the least amount of work necessary and update the contents of location X. This is what happens in most languages with this probelm. Therefore, to update the referenced location it is necessary to write the second assignment statement as "val X: = 5". Then the Lvalue which is the contents of X is updated. Using tuples in PAL the program becomes

let X=loc 2 in

X: = nil aug loc 3;

X 1: = 5

It is easy to see that except for the inconvenience of creating a 1—tuple there is little difference between the two languages. They both have the problem of distinguishing which location is to be updated.

An analogous problem occurs in defining equality for locations. In S-PAL two locations are equal if and only if they share. This corresponds to equality defined by the eq predicate in LISP. However for arithmetic operations, it is desirable to define two locations holding the same value as being equal. This corresponds to the equal predicate in LISP. The distinction between these two definitions is discussed at some length in Park [26]. The S-PAL definition was choosen because locations are values in S-PAL and the polymorphic operator "=" is defined over all other values. The affect of equal can be achieved by using val to extract the contents before equality is tested. However, the need for two approaches is inherent in the concept of location.

Dynamic Variation of Bindings

S-PAL like GEDANKEN, CPL and other languages requires that once a variable is bound to an object that binding is fixed for the duration of the execution. However, BASEL[10,12] allows the programmer to vary the bindings of variables dynamically. The reason for this appears to be connected with the concept of "type" found in BASEL. Both variables and locations may have associated types. A typed location may hold any value which is consistent with the type. A typed variable may be bound to any object which is consistent with the type. Suppose X is a variable which can either be a location of an integer (loc int) or a location of a real (loc real). Then, at any time X may be bound to a loc int or a loc real but not both. That is, if X is a loc int, then the assignment X: = 3.141 will fail. If both types of values should be assignable to X, then X should be of type loc union (int, real) and in that case X is bound to a location which can hold either integers or reals. (union lists alternative forms) Then either X:=2 or X: = 2.7 is a legal assignment. Allowing variable bindings makes the distinction between locations and binding a little more obvious. These topics are discussed again in context of types in Chapter V.

The most obvious affect of this alternative is to increase the amount of confusion a computer must handle. It becomes difficult to insert type validity tests for variables if the binding is unknown. Since it is in general impossible to predict program flow, it is necessary to assume the worst and test for the type of object to which the variable is bound. This is unnecessary if bindings are fixed since the type is determined when the variable is defined and bound.

Variable bindings also affect how the processing of free variables is done. In BASEL the free variable list is built from the values currently bound to free variables. Hence, any future rebindings will not affect the values of the free variables when the function is applied. However, in PAL where the free variable list is kept by name, a rebinding would affect the value obtained in future function invocations.

Chapter III

Representing Data Structures by Functions over Symbolic Domains

The only tool for building data structures in the current PAL is the tuple. The major properties of the tuple were discussed in the previous chapter in connection with locations. The tuple is a perfectly general device for building and referencing collections of data. Therefore, any new technique for data structuring will not expand the capabilities of the language. However, the tuple is a "natural" representation primarily for data which has some order to it. That is, there is a natural integer index associated with each data element. This data may be a vector of points, a string of characters, etc.

Representing Data without a Natural Ordering

When the data is without a natural ordering, as is the case in a number of data collections, the tuple is a much less attractive form of representation. Consider for example the representation of the control items in the gedanken interpreter for PAL. It is possible to represent these elements as tuples but it is awkward because many conventions must be introduced.

For example, the control item for a λ-closure has three components which can be succinctly described in the notation of Landin's [13] structure definition as

A λ-closure has
    a bound_variable_part
    and a λ-body
    and an environment.

When this is translated into a tuple representation, it is necessary to establish conventions such as the first component will be the bound_variable_part, the second component will be the λ-body, etc. Furthermore, it is necessary to be able to recognize the type of the control item so an additional convention is required to store the type information. Thus, a λclosure might be represented (as it is in R-PAL) by the following set of definitions

<u>def</u> Is λclosure X =
    Istuple X → X 1 eq'λ' | <u>false</u>
<u>and</u> BV X = X 2
<u>and</u> Body X = X 3
<u>and</u> Env X = X 4

The structure definition is simpler because only the necessary information is supplied. Irrelavent information such as the order of the components is not needed. Thus, the tuple definition suffers from overspecificity: it is necessary to stipulate conventions which are not strictly required to define the structure.

Obviously, this is only one of a number of possible representations in terms of tuples. Other representations may be used to make the processing of the data structure easier. For example, in the abstract syntax of PAL [40] the structure type is represented as the last component of the tuple. Another variation is used in the representation of control items in GEDANKEN [30]. However, in all these representations in terms of tuples or vectors, the definitions have more structure than is needed.

Difficulties with Tuple Representations of Data Structures

One of the most unnatural aspects of tuple representations of data structures is the handling of the structure type information. This is most often represented by a tag which is stored in a standard location in the structure and identifies the type or class of the structure. Since it is part of the tuple it becomes necessary to program around it for various actions on the tuple. For example, the tag is the final component in the PAL abstract syntax structures. Hence it must be removed and replaced whenever the tuple is augmented.

Another unnatural aspect of using tuples is that they have too many properties. It is impossible to restrict action on a data structure only to operations applicable to that data structure. Since it looks like a tuple, it can be manipulated as a tuple as well as the data structure it represents. This leads to confusing programs. It also inhibits optimization which depends on the structure since all the tuple properties must be preserved whether or not they will be used. The tuple gives a <u>weak representation</u> of the data structure. It has the properties of the data structure and also its own tuple properties. To have more control it is necessary to have a <u>strong representation</u>. That is, a representation which has only the properties of the data structure and no others.

<u>The Properties a Data Structuring Facility Should Possess</u>

The above discussion indicates a set of properties which a data structuring extension should have to be more natural and convenient.

1) The representation of the data structure should be strong to allow optimal storage and to reduce confusion.

2) The type of a data structure should be easily accessable and independent of the data in the structure.

3) It should be possible to access the data using its natural identifier.

In addition to the above properties the data structuring capability should be convenient to use. This means that the syntax should be relatively simple, not too verbose and in general natural to read and write. It should also, if possible, provide documentation on the attributes and form of the data structure.

The facility should also provide a number of different ways to build data structures. In some problems it is impossible to predict the form of the data structure and it must be possible to construct it dynamically. This type of data structure is available in languages like LISP [21], ALGOL68 [37] and is discussed in a number of papers, in particular that of Hoare [11]. The dynamic form is perfectly general but there is a real cost associated with constructing and storing the data structure.

For some problems, such as payroll management, it is possible to define a fixed format for the data. In this case the relationship of data items is not varied during the processing. Therefore, it is possible to optimize the storage and processing of such data structures. COBOL [36] is typical of languages which provide this static data structuring capability. Obviously these two forms are extremes and a general purpose facility should allow a wide range of possibilities between these forms.

Landin's Structure Definition

A modified form of Landin's structure definition was chosen as the basis for the data structuring facility which we shall discuss. There were two reasons for this. First it satisfies many of the above goals. Secondly since many of the ideas of PAL were derived from Landin's ISWIM[17], it appeared that the structure definition syntax would fit in well with the rest of the PAL syntax. It is not yet clear how well the actual formalization of Landin's syntax meets such goals as simplicity and naturalness. Only actual use will be able to resolve these questions.

What features are needed in a facility for structuring data? This question is discussed at some length in Landin [13,16]. Only the conclusions will be reproduced here. If you have a data structure it must be possible to recover the individual data items which make up the structure. Therefore, there must be a set of <u>selectors</u> which can be used to extract the data items. Conversely given a set of data items it must be possible to build a data structure whose components are that set. Thus a <u>constructor</u> which takes sets of data items into data structures is required.

Finally when processing a data structure it must be possible to distinguish between alternative forms of that structure. For example, a component of a structure might itself be one of several data structures or primitive values. Which form occurs can be determined using a set of predicates for the alternative types. Each predicate is a function on the universe of discourse which yields true whenever its argument is of the specified type. Therefore, the structure definition must provide at least enough information to define

1) a set of selectors

2) a constructor

3) a predicate

An Additional Property of Landin's Structure Definitions

Actually Landin's definitions provide slightly more information than we have discussed thus far. Our earlier definition of a λclosure provided only enough information to define the selectors and the predicate. A more complete definition of λclosure would be

A λclosure has

a bound variable_part which is a variable

and a λbody which is a λexpression

and an environment which is an environment.

The difference is that now each component also has a type associated with it. This makes it possible to check the type of each component before the data structure is constructed.

This makes it possible to provide a stronger representation than is possible without the type information. It prevents unexpected data from occuring in the structure. If any data item is allowed as a structure component it is impossible to restrict the properties of the data.

The addition of type information for the components complicates the description of the structure definition. Further discussion of the problem involved is therefore delayed until Chapter V.

Other Formalizations for Data Structures

Data structures have been formalized by several methods. A good commentary on previous formalizations is given in Standish [33]. He presents a method which is similar to Landin's structure definition but has a more concise syntax. In recent work Vigor [38] proposed a definition which included the selectors, constructor, and predicate, and also added some functions to force different modes of evaluation (applicators) and to change representations (designators). Similarly, Burstall and Popplestone [4] add an inverse (destructor) to the constructor which produces the components of the object.

Another approach to formalizing data structures is to represent them as graphs. These graphs have nodes which represent the structures and the edges of the graphs represent the relationships between the structured objects. AMBIT/G[5] and VERS[7] are typical of languages which use this approach.

In the case of VERS the graphical form must be converted into a machine representation by using a set of primitives for manipulating the structure. The primitives are machine independent and are derived from operations for constructing and manipulating the graph. Efficiency is obtained by substituting different machine oriented definitions for the primitive operations. That is, a single primitive may have a different implementation for each structure type. This makes it possible to tailor the primitive action to the manner in which the data will be used. This idea of defining "code" to implement a particular instance of a primitive is also present in the work of Laurence [19]. Machine independence still exists since it is only necessary to redefine the primitives for the new machine. The structures are coded in terms of the primitives so they are unchanged.

Unfortunately, the primitives that are used in VERS seem to force a particular form of implementation. It appears that all data structures must be created and linked dynamically at run time. This makes it impossible to group several substructures into a single major structure with fixed links and then use the fact that the link relationships are fixed to optimize references to components of the substructures. This type of optimization is seen in PL/I and COBOL where components of substructures can be given fixed offsets from the address of the major structure. One advantage of the structure definition is the lack of commitment to any particular implementation.

The S-PAL Representation of Data Structures

The formalization of data structures should be chosen to maximize implementation independence. That is, formalization which unnecessarily restrict the implementation should be avoided. If this were the only requirement on the formalization, then the only way to avoid introducing extraneous restrictions would be to axiomatize the desired properties. However, it does not seem possible at this time to develop a meaningful set of axioms which fully characterize a datastructure.

Axiomatization also makes it difficult to build on previous definitional work. There is a definite pedogogical advantage in defining new features in terms of the existing language structure. This reduces the amount of work needed to relate the new features to the rest of the language. Part of the design philosophy of PAL was to develop the language in several "logical bootstrap" operations. In each step the new features were formalized in terms of the language defined in the previous step.

We have chosen to formalize data structures in S-PAL in terms of a specific R-PAL representation. Although this is more restrictive than is theoretically necessary, we believe the pedogoical advantages outweigh the other costs. A structure definition is basically a description of a labelled node in a directed graph with label edges. Hence the choice of syntax has already restricted the set of possible representations. The representation which will be used was chosen because it appears to add very few additional constraints to implementing the structure definitions.

The process of formalizing data structures in terms of the chosen representation can be divided into four parts.

1) Defining a syntax in which it is convenient for the user to define, create and manipulate his data structures (the concrete syntax)

2) Defining an abstract representation for the information contained in the above syntax (the abstract syntax)

3) Describing the translation of the syntactic information into a representation of the data structure (the interpretation of the parse or the standardization process)

4) Presenting the properties of the chosen representation (semantic clarification)

The approach to part 1 has already been discussed. We continue the description with an informal discussion of part 4 because it is basic to the other parts of the formalization process.

<u>A Functional Data Structure Representation</u>

In Landin's approach, data structures are treated as a new class of constructed objects. The predicates and selectors are functions whose domain includes these objects. In particular a selector returns a component of the data structure as its value. In S-PAL data structures are instead represented by a special class of functions called data functions. These functions are defined over the set of selectors for the data structure. A component is obtained by applying the data function to the selector.

To make the distinction between the two forms of representation clear, consider the functionality (i.e., domain and range) of the traditional [4,13,33] form

predicate ε objects $\rightarrow$ truthvalue

selector ε data structure $\rightarrow$ component

constructor ε set of components $\rightarrow$ data structure

In the S-PAL representation the functionality is

constructor ε set of data components $\rightarrow$ data functions

data function ε selectors $\rightarrow$ data components

predicate ε objects $\rightarrow$ truth value

This approach of using functions for representing data is not original. It is used in Gedanken[30] and by Park[26] and Balzer[1]. However this formulation differs in several aspects from their approaches.

The functional approach is a natural generalization of the tuple. In the tuple the constructor is aug, the selectors are integers and the predicate is Istuple. To get data functions we extend the domain (selector set) to symbolic names so the components of a data structure may have descriptive selectors. The constructor will build a more general class of functions and a whole set of distinct predicates will be created.

Data Functions Provide Flexibility

The reason for choosing a functional representation is the flexibility it gives to program construction. The program can be written with functions representing the data. Then when the algorithm is clear the functions defining the data structures can be written in a form best suited to the way the data is used. For example a sequence of elements can be represented as either a list or an array depending on how the data will be referenced and manipulated. The important point is that it is possible to change the representation whithout changing the algorithm.

The user may choose to use the functional representation created by the translation of the structure definition or he may define his own function to represent the structure. In the latter case it is possible to choose the representation to suit the problem. For example, it is possible to define the values of a subset of the components in terms of the values of the other components. Then it is necessary to store only the independent components in the environment of the function. The dependent components can be calculated from the stored values. This is a way to save storage when the components are related and it illustrates one of the possible ways a data structure can be varied within a functional representation.

Consider for example a collection of data indexed by the integers from 1 to N in which the values on the odd integers are the squares of the values on the even integers. If we assume that there is a function Evendata which holds the values of the function for even integers, then the whole collection can be represented by

```
def Datacoll X =
        Odd X → [Evendata((X-1)/2)]**2
              | Evendata(X/2)
```

Thus, only the even values need be stored. The functional form of representation makes it easy to replace the data with an algorithm which calculates the data.

Atoms

The integers make very good selectors. They can be computed, they are ordered and their meaning does not vary from occurrance to occurrance. To extend the domain of data functions, it is desirable to use symbolic selectors with properties similar to the integers. There is no strong argument for being able to compute symbolic selectors and we have noted earlier that an ordering of symbolic selectors is not important. Therefore, the only property of an integer which is important for symbolic selectors is the invariance of its interpretation. For example, the numeral 2 always designates the integer 2. The designated value does not depend on the context of the designation; in other words it is a constant.

The idea of invariance is important because a data function is given only the value of the selector to use in selecting a component of the represented structure. If the same selector designation specified different values in different contexts then applying a data function to what appeared to be the same selector could produce different results depending on the context in which it occured. Therefore, it should be possible to designate a symbolic value in a manner which does not depend on the context of the designation. An example of such a designation is the character string constant found in PAL and many other languages.

Although a string constant satisfies the invariance property it is not completely suitable for use as a selector. The reason for this is that strings have too many properties. The only property a symbolic selector must have is that it must be possible to test any two symbolic selectors for equality. This property is used in the data function to identify which component is being selected. However, character strings have many additional properties such as the ability to be concatenated, decomposed, etc. This means that any representation of character strings must preserve these properties. On the other hand if equality is the only property required of symbolic selectors it should be possible to use an encoded representation. For example, they might be represented by a type code (tag) and an integer identifying the selector. This representation uses much less space than a full character string and is much easier to manipulate on more computers.

To take advantage of the simpler representation requirements of the symbolic selectors, a new class of objects called atoms is introduced. Axiomatically their properties are

AxIII.1) The class of atoms is distinguishable from all other objects in the universe of discourse.

AxIII.2) Any two atoms are either testably equal or distinct.

AxIII.3) There are no other properties.

Because atoms are normally represented in an encoded form it is necessary to specify how the correspondance between the external designation and the encoded internal representation is established. To preserve invariance this correspondance should be 1-1 and should depend only on the external designation. In most implementations this is accomplished by encoding the atom by a type code and the address of a copy of the external designation. Therefore, the character string for the external designation need be stored only once. If this copy of the external designation is unique, then any occurance of the atom in its external form can be uniquely converted into the internal representation. Conversely, each occurance of the internal representation uniquely identifies the external designation. Therefore, the correspondance is 1-1.

Alternative Definitions of Atoms

The atoms defined here differ from the atoms defined in both LISP[21] and GEDANKEN[30]. In GEDANKEN the atoms are objects without an external designation. They only have an internal representation which consists of a type code and an integer value. There is a primitive operation which generates new atoms whose identifying integer is distinct from those of all previously generated atoms. In this case the representation of the atom has no significance other than to distinguish different atoms.

To use these atoms for selectors it is necessary to give them identifiers which can be used as external designators. This is accomplished by binding names to the atoms used as selectors. However, this approach does not provide invariance of selectors. It is possible to bind the same name to two different atoms in different contexts. Therefore, it is possible to have two atoms with the same designation which are not equal. In addition, this approach does not allow atoms to be output on a printer or a removable storage device because there is no external designation. This also means that an atom cannot be referenced by name in another program using the same data base.

In LISP there are both named and generated (unnamed) atoms. But these atoms have too many properties for our purposes. Each LISP atom also represents a value. The value is stored with other descriptive information in a property list which is attached to the atom. This is a list of attribute and value pairs. The only LISP attribute that an S-PAL atom has is its external designation or print name. This is determined by the 1-1 correspondance between atom values and names so there is no need to have a property list. Although S-PAL atoms are more primitive than LISP atoms, it is possible to define an S-PAL data structure which represents the LISP atom if the additional properties are needed.

In LISP all names are atoms so there is no problem with syntactically distinguishing the atoms. However, in PAL names which are not atom names already exist. In fact, since S-PAL atoms do not have associated values, non-atomic names are necessary to identify locations and other objects. Because selectors will be used fairly frequently it is desirable to have a convenient and easy syntax for atoms. This is another reason why character strings were not used as selectors. Quotes are too cumbersome, especially for short names. Several different schemes were proposed of which the best appeared to be to use strings of two or more capital letters or numerals with at least one capital letter. This seemed more convenient than using a special marker such as the quote in a character string. It does, however, mean that names which were previously available for variable identifiers are no longer usable for that purpose. Thus, existing PAL programs may be invalidated.

The Special Properties of Data Functions

Why is it necessary to identify a special class of functions to represent data? The main reason is that an unrestricted function has too many properties so that it is possible to build an efficient representation and so that some basic questions about the function are decidable.

An example of an undecidable question for general functions is what is the domain of definition of the function. However all data functions have finite domains. In the case of tuples it is possible to find the entire domain with the Order function. This allows the user to write algorithms which process every element of a tuple by sequencing through the domain of the tuple. This property is also necessary for symbolic domains. For example, a user might test two instances of a data structure for equality by applying the two functions to each of the possible selectors and comparing the results. Obviously he must know the selector set to do this.

The Order function is applied to a tuple to get the domain information. However, it is as we noted above impossible to extract that information from an arbitrary function. Therefore, it seems more natural, following the approach used by Reynolds [30], to require a data function to produce its domain when it is asked.

It is not feasible to predict every question which might be asked about a function so we will restrict our attention to questions which appear to be useful for manipulating data structures. This relatively small set of questions can be encoded by a set of special selectors which are recognized by all data functions. These special selectors will be designated by built in atomic keywords (e.g., domain). These are built-in constants just like true or false.

This leads to a natural definition of a data function. A _data function_ is any function whose domain includes the set of special selectors and which gives correct information about the function when applied to those selectors. Note that this definition makes it impossible to decide if an arbitrary function is a data function. However, this is less important than the fact that a user may define his own data functions if he so chooses. He need only check for the special selectors and produce the correct results. Such user defined functions will be operationally indistinguishable from the functions produced by structure definitions.

The Selector Set

The result of applying a data function to the special selector _domain_ is a tuple consisting of all the selectors in the domain of the data function. Because there is no way to compute the selectors from a smaller amount of information it is not possible to produce an abbreviated form of the domain information such as that given by the Order function. The only complete specification is the set of atoms themselves. The special selectors are not included in the tuple produced in response to _domain_ because all data functions are assumed to be defined on these selectors.

Predicates

The predicates are functions which extract a different type of information from the data functions. Since the predicate will most often be used in a functional context it is unreasonable to replace the predicate with a selector. However, it is reasonable to require the function to produce information which the predicate can use in deciding if its argument is of the correct type. Defining the type of an object is a very complex subject. The most natural definition of two objects having the same type is that they can not be distinguished (except for values) within the language. This definition is, however, impossible to implement. Therefore, S-PAL has left the decision on type equivalence up to the user. But we provide facilities which the user can use to build a type predicate.

One way to type a data object is to attach to every instance of the data object a tag which identifies that object. Hence, a primitive definition of type equivalence is that two objects are the same type if they have the same tag. This means that it is possible to have two data structures which would be operationally equivalent, but are not considered equal because the tags differ. The loss of this equivalence capability is a small price to pay for the simplification it provides in handling types (see Reynolds[31], Morris[25]). The tag can be viewed as a characterization of the data structure in a single symbol.

It should be noted that the tag is often insufficient to characterise the structure fully. For example, a user may want to consider two arrays to be of the same type if and only if they have the same dimensions and bounds. This means that it is necessary to use the domain information to fully validate the type. Another approach to type equivalence is that two structures are equivalent when they are constructed by the same constructor function. However, both of the alternatives imply that the tag information is identical.

Because the tag appears to be the most primitive form of type information it is the sole attribute that will be tested by the predicates which are automatically generated by the translation of the structure definitions. If the user wants to define more complex type tests he can program them. The predicate function extracts the tag information by applying the data function to the special selector tag. The result is the atom which was used to tag the structure. This can then be compared against the tag expected by the predicate and the result of this comparison is the result of the predicate.

The Constructor and Constructed Objects

Since the data structures are represented by data functions in S-PAL the constructors are function producing functions. A constructor function is automatically generated for each structure definition. It takes as its only argument a tuple whose components are the components of the data structure. The order of the components in the tuple determines the selector with which they are associated. The selectors are ordered by the order in which they occured in the definition. The selectors and tuple components are then paired in their order of occurrence. The result of applying the constructor is a function which will produce the appropriate component of its argument when it is applied to a selector in its domain.

There is a special selector constructor which will produce the constructor of a data function. As we noted above this is useful when defining the type of an object. However, there is a more important reason for including this as a special selector. In some applications it may be necessary to change a component of a structure. If the component is an Lvalue there is no problem. If, however, the component is an Rvalue it cannot be replaced by assignment. Therefore, the only alternative is to build a new copy of the structure with the component replaced.

This is only possible if the constructor which was used to build the original function is available. If it can be determined solely from the data function, then it is possible to write a general purpose update function. This function would take a data function, a selector, and a value as arguments and would return an updated data function. The result could be computed by constructing the tuple of components of the data function using the selector set. Then the appropriate component could be replaced with the new value. Finally a new copy of the data function is produced by applying the constructor obtained from the original data function to the new tuple of components. This function is used in Chapter V and justifies the inclusion of the constructor selector.

Universal Constructors

If one of the major uses of constructors were in rebuilding data functions, it might be simpler to have a universal constructor function which took as an argument the type of function to be constructed as well as the components to use. This universal function would look up the type and build the data function corresponding to that type from the components. This way the special constructor selector would not be needed because tag would provide the required information. This approach is developed in greater detail in the formal definition of GEDANKEN.

There are, however, several disadvantages to this approach. First if atomic types are used it is necessary to search the entire list of all defined types to find the information for constructing a representation of a particular type. This could be very inefficient although hashing techniques might help. Also because there is a need to keep this list of defined types, dynamic declarations of new types are more costly since each new definition makes the list larger. Another problem is that the atomic tag may not uniquely identify the type of the structure. It might be necessary also to include the selector set in the arguments to the universal constructor.

Thus, it appears that a universal constructor is only practical if the argument which specifies the form of the structure contains all the information necessary to build the data object. This approach was used by Standish[33]. He defined data descriptors which are Rvalues which encode the description of the structure. Then to build a structure there is a constructor which takes a set of values and a descriptor and produces the constructed object. However, it appears that it is better to build the constructors directly since in that case it is possible to optimize them for the particular data structure they are building. The S-PAL constructor is analogous to Standish's descriptor because it must contain all the information necessary to build the constructed object.

We complete the informal description of the desiderata for the data functions with a few comments on two of the special modifiers Standish introduced into his descriptor definition. These modifiers are used to attach additional attributes to the structures. The predicate modifier makes it possible to add an additional predicate function to the predicate generated automatically. The generated predicate then yields <u>true</u> if and only if the structural properties are satisfied (e.g., correct tag) and the modifying predicate is also satisfied. This appears to be strictly unnecessary since it is always possible to include the generated predicate in a user defined predicate which has the additional tests.

The constructor modifier is a function which is invoked after each construction and can be used to initialize values in the constructed object. For example, it can be used to close a ring of pointers which can not be done with a purely functional description. Like the predicate modifier this effect can be achieved by including the constructor in a function which uses the constructor, then performs the initialization on the result. Because these modifiers can be easily programmed in S-PAL they will not be included in the structure definitions defined in the next chapter. They are primarily useful abbreviations.

Chapter IV

The Formal Definition of Data Functions

Simple Structure Definitions

Landin [13,16] used an informal syntax for structure definitions and for naming the various functions the definition produced. Selectors and predicates received explicit names while the constructor name was derived from the predicate name. A slightly different approach is used in S-PAL. The selector names are all explicit in the definition and are given as atoms. The definition for λclosure would be written as

def LAMBDA_CLOSURE which has

BND_VAR_PART

(1) also LAMBDA_BODY

also ENVIRONMENT

This definition is intended to define a constructed object or data structure of type LAMBDA_CLOSURE. "LAMBDA_CLOSURE" is an atom and would be the result of applying an instance of the data function to the special selector tag. The names for the predicate and constructor are derived from the type by using the prefixes "Is" and "Make" respectively. For example the above definition would yield the two functions IsLAMBDA_CLOSURE and MakeLAMBDA_CLOSURE.

The eventual goal of the above definition is to define two functions, the constructor and the predicate. It would be possible to write this in the form

(2) def IsLAMBDA_CLOSURE = ...
and MakeLAMBDA_CLOSURE = ...

where the ellipsis represent function definitions. However, we have chosen to use the structure definitions as syntactic sugaring for the above forms. Therefore, it will be necessary to define an abstract syntax for structure definitions and to expand the standardizing section of the gedanken interpreter to convert the abstract structure definitions into the desugared form.

The Syntax for Simple Definitions

Since the results of a structure definition is to be definitions like phrase (2), it is natural to extend the class of <basic definitions>(D3 in the abbreviated syntax). Hence, D3 becomes

D3:: = NAME{,NAME}$^{\infty}_{0}$=E | NAME V=E
| (D) | [D] | S

Where S stands for <named-structure>. The elementary structure definition syntax is

<named-structure>::=ATOM which has <anonymous-structure>
(3) <anonymous-structure>::={<selector> also}$^{\infty}_{1}$ <selector>
| only <selector>
<selector>::=Atom

In abbreviated form this becomes

S::= ATOM <u>which</u> <u>has</u> S1

S1::= $\{S2\ \underline{also}\}_1^\infty$ S2 | <u>only</u> S2

S2::= ATOM

The interpretation of the syntax is as follows. The <named structure> gives a tag to a collection of components given by the <anonymous structure>. We shall see that the <anonymous structure>can occur elsewhere in the syntax, so it must be a recognizable syntactic entity. Therefore, it consists of either two or more components separated by <u>also</u> or a single component prefixed by <u>only</u>. Each component is a selector specification which is an atom.

The abstract syntax for the above concrete syntax will be represented pictorially and by R-PAL programs following the method established in Wozencraft and Evans [40]. Figure 1 shows the graphical abstract syntax for (3). The abstract syntax tree for definition (1) is given in figure 2.

<u>The Primitives for Defining the Constructor and Predicate</u>

The next step in the processing is to build a standardized definition like definition (2). The details of this process are delayed until later in the chapter. The expected form is a simulataneous definition whose righthand side defines functions representing the constructor and predicate. These two functions are defined by two new primitive functions,

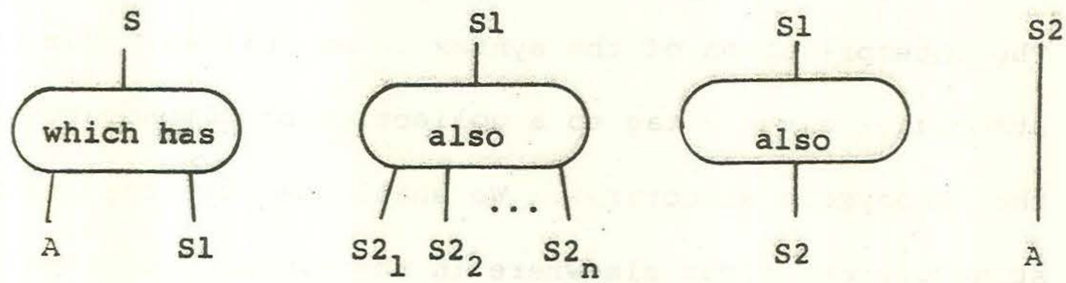

figure IV.1 Abstract syntax for simple structure definitions

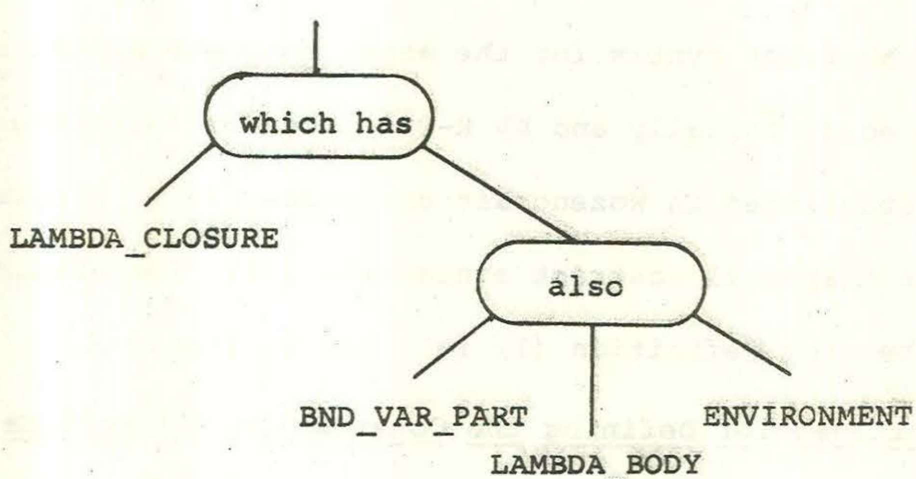

figure IV.2 Typical abstract syntax tree for a simple structure definition

MakeStr and IsStr. These constructor and predicate building functions take the tag and selector set information and produce functions with these parameters as "own" variables. These functions are then bound to the constructor and predicate names when the definition is evaluated. The standardized tree for definition (1) is shown in figure 3.

The functions MakeStr and IsStr could be defined in terms of R-PAL in a manner similar to that used by Standish [33] to define a constructor given a description of the structure. The main reason this is not done is that defining the constructor and predicate in terms of a primitive function allows more flexibility in the implementation and hence greater efficiency. Before describing a representation of these functions, it is useful to expand the syntax of structure definitions.

Predicates for Types with Alternative Forms

It is often the case that a particular structural type will occur in several different forms. For example, following McCarthy [22] we can define the abstract syntax of a term in an expression as

```
        def TERM which
            is (SUM which has
                    ADDEND
(4)                 also AUGEND)
```

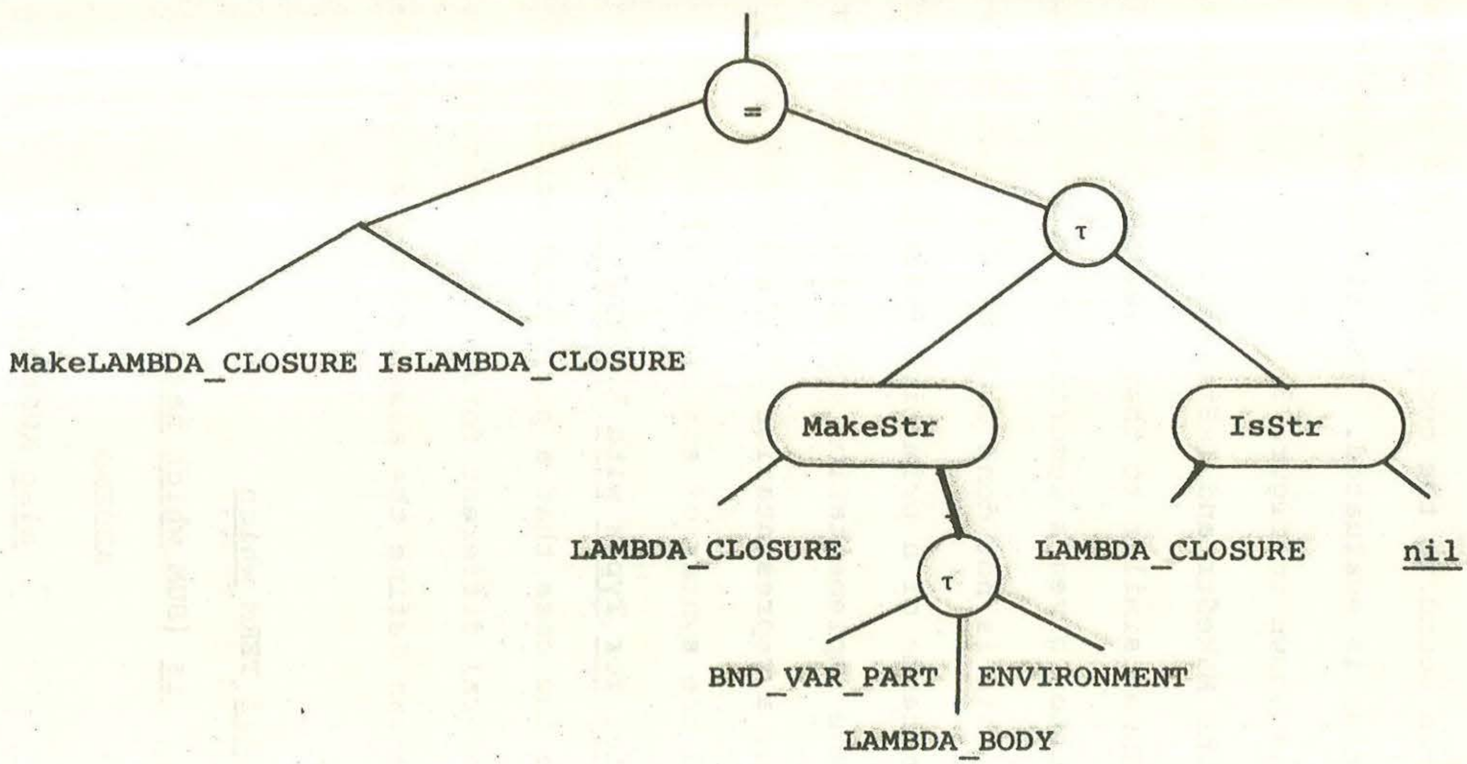

figure IV.3 The standardized form of the syntax tree for definition (1). It is composed of a simultaneous definition of the constructor and predicate which result from the application of MakeStr and IsStr respectively.

else is (PROD which has

MPLIER

also MPCAND)

else IsCONSTANT

else IsVARIABLE

In this definition a TERM can have four structural variants. It can be one of the two constructed objects SUM or PROD or it can satisfy one of the previously defined predicates IsVARIABLE or IsCONSTANT.

The Differences Between Predicate and Structure Definitions

There are several important facts to notice about this definition in contrast to the previous definition. First the definition begins with which* instead of which has. The phrase which designates that the type being defined is not a constructed object, but instead defines a class of constructed or elementary objects. That is, there is no way to construct a TERM. It is only possible to construct the two variant forms SUM and PROD. Therefore, there is no constructor associated with a which definition. Only the predicate which recognizes members of the class TERM is defined.

The various alternatives in the class TERM are separated by the connective else. These alternatives may be constructed objects such as SUM, or elementary or previously defined

---

*The is following which is a noise word which improved readability.

objects such as IsVARIABLE. If an alternative is a constructed object, a constructor and a subpredicate must be defined. This is indicated by the _which has_ which designates that the type to its left is a constructed object. Thus the phrase

SUM _which has_

ADDEND

also AUGEND

defines a constructor and predicate just as if it appeared alone in a definition. The parentheses are necessary to make clear the scope of the alternatives.

Thus, we see that the two forms of structure definitions have different purposes. The _which has_ form defines both a constructor and a predicate from the set of component selectors. The _which_ form defines a new class predicate from the set of predicates which are alternative forms of the class members. We note in passing that it was necessary to use _also_ and _else_ instead of the more natural _and_ and _or_ used by Landin because _and_ and _or_ already have meanings in PAL.

The Syntax for Predicate Definitions

To add predicates to the syntax it is necessary to modify D3 once again.

D3::=NAME{,NAME}$_{0}^{\infty}$=E | NAME V = E

|[E] | (E) | S | P

where P represents a <named predicate>. The syntax for simple predicates is analogous to that for simple structures.

<named-predicate>::=ATOM which <anonymous-predicate>

<anonymous-predicate>::=<predicate designator>{else
<predicate designators>}$_{0}^{\infty}$

<predicate-designator>::=<function>|is(<named-structure>)

(5) or in abbreviated form:

P::= ATOM which P1

P1::= P3 {else P3}$_{0}^{\infty}$

P3::= P2 | is (S)

The corresponding abstract syntax is given in figure 4, and figure 5 is the abstract syntax tree for definition (4). The interpretation is that a <named predicate> defines a class from an <anonymous-predicate> which is a list of alternative <predicate-designator>s or predicates from <named-structure>s.

The standardized form of definition (4) is more complex than that of definition (1). The problem is that not only is a predicate being defined but so are two constructed objects and their associated constructors and predicates. Therefore, the lefthand side of the simultaneous definition has become a tree of functions. This form is similar to that which occurs

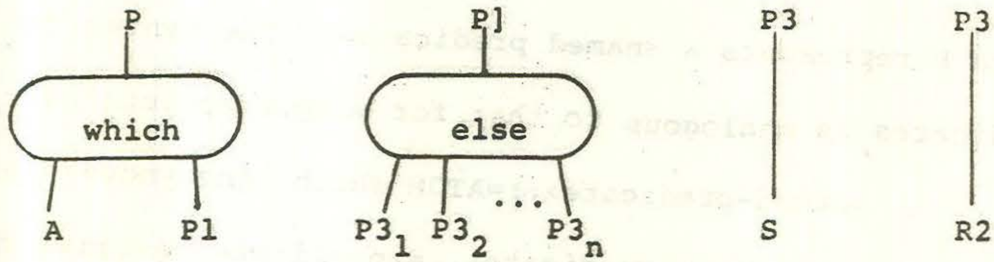

figure IV.4 Abstract syntax for predicates

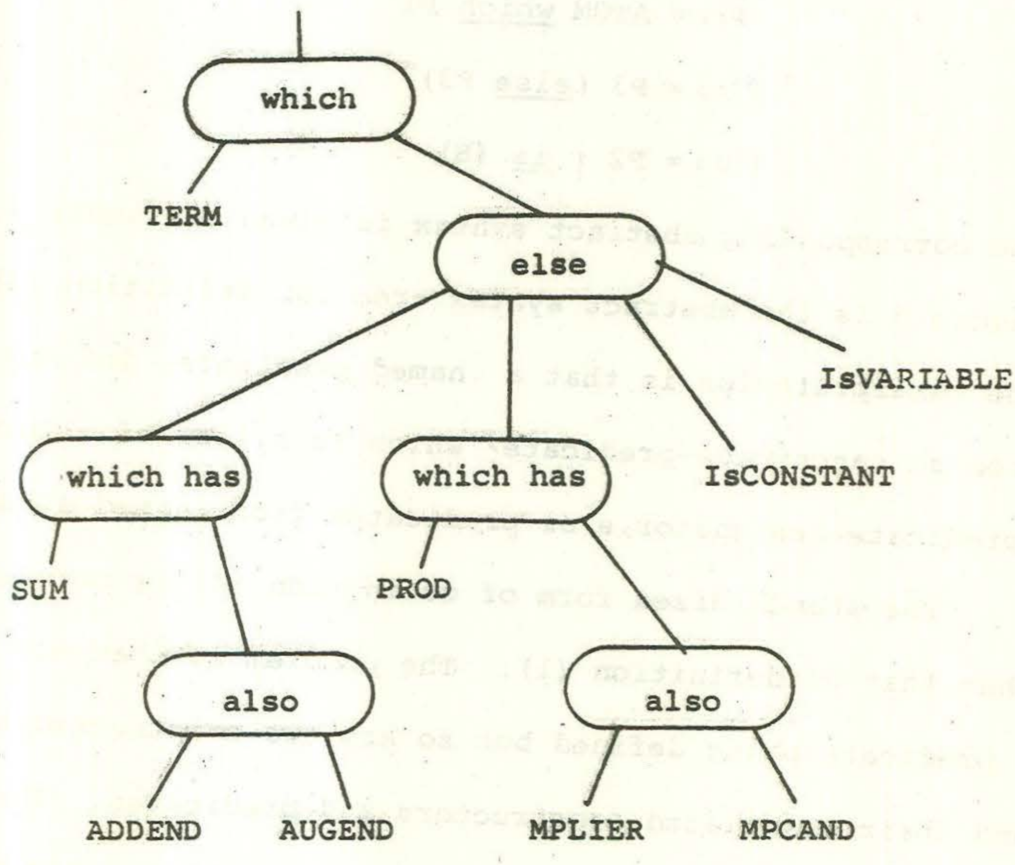

figure IV.5 Typical abstract syntax tree for predicate when definitions are nested in the current PAL. The standardized form of (4) is given in figure 6.

A Representation for the Primitives MakeStr and IsStr

It is now time to describe the primitives for building the constructor and predicate. This will be done by showing a representation for the data function in terms of an R-PAL program and indicating how the constructor (predicate) for that data function is built. An R-PAL representation is used to show data structures are basically applicative. The previous chapter gives a set of properties the representation must have.

1) It must provide a type indicator such as a tag
2) It must be able to generate the selector set
3) It must provide its own constructor
4) It must be able to produce the data component corresponding to each selector

These constraints can be satisfied by using a function which has available as own variables the selectors and the type, and stores the data components in a tuple. A component is selected by searching for the selector in the selector list and finding the index of the component in the data tuple.

This is certainly not the only possible representation of a data function. There are many other possible representations. The reason this approach was chosen is that it takes advantage

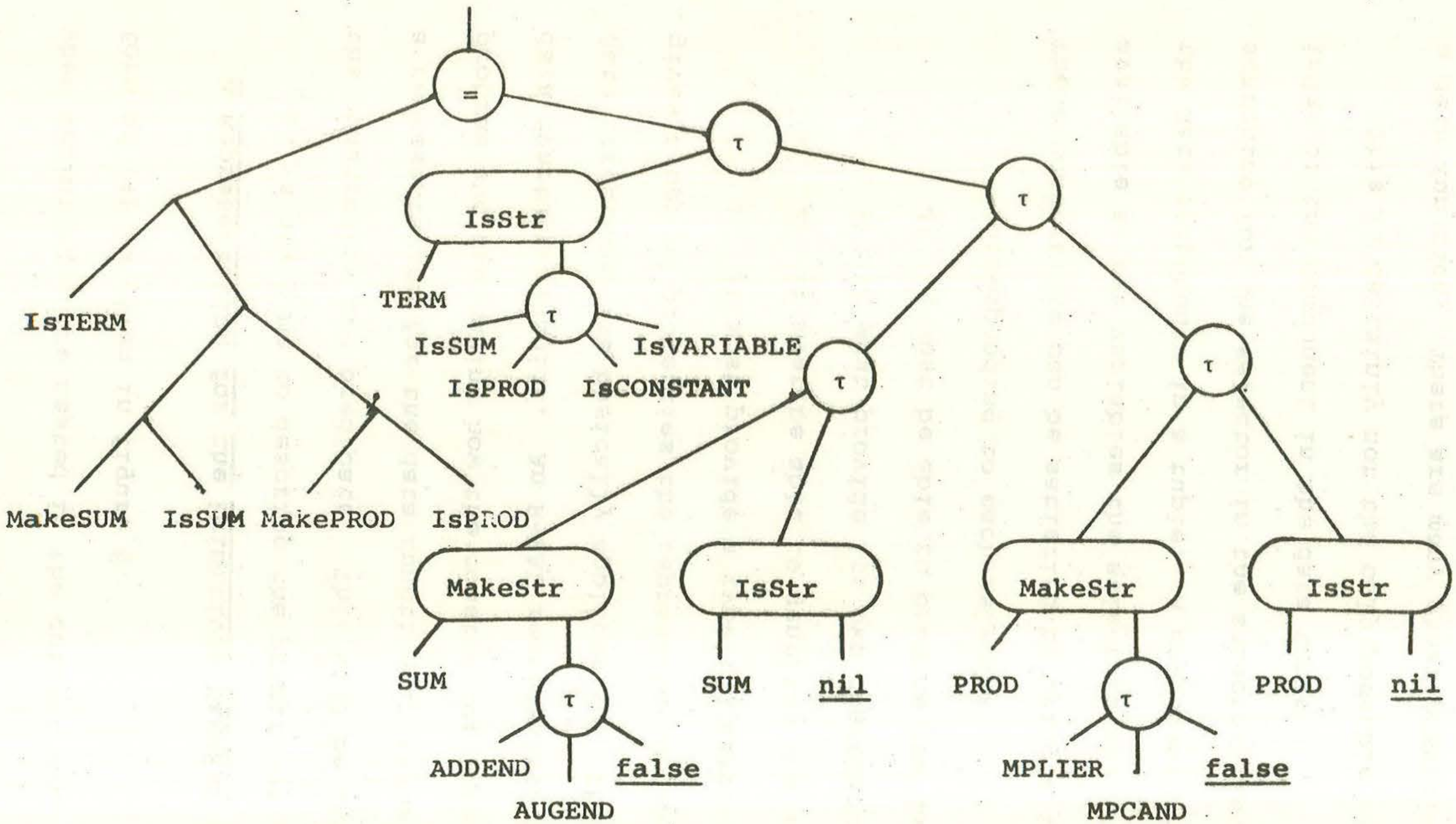

figure IV.6 The standardized tree for definition(4). An extra selector indicated by false has been added to indicate the selector set is fixed. This will be explained under mixed domains later in the Chapter.

of the powerful data representation properties of tuples and has very little extra complexity. Furthermore, the conversion of a selector to a tuple index can be speeded up by hashing the atom name to get a tuple index. The particular hashing scheme may depend on the data structure to get a 1-1 correspondence between atoms and tuple indices.

The Predicate Building Function IsStr

A predicate is defined from two sets of data. First there is the tag by means of which the data function describes itself. Secondly, there is the list of predicates for alternative types which define a complex predicate. The function IsStr takes these two arguments and returns a predicate function. This predicate function tests the validity of its argument by first applying the set of alternative predicates. If any of these yield _true_ then the predicate function yields _true_. If none of the predicates yield _true_ then the tag of the argument is compared against the tag built into the predicate. In this case the result of the predicate is the result of that comparison. The R-PAL representation of IsStr is given in figure 7.

Throughout this thesis, definitions and representations will be written to emphasise their structure rather than to

```
def  IsStr (Name,Predicates) =
        { fn y. [ Istuple Predicates - >
                      ( Q (Order Predicates)
                        where rec Q k =
                            k eq 0 -> false
                                | Q (k-1) or Predicates k y )
                    | Predicates y ]
                 or ( y tag eq Name )
```

figure IV.7 The representation of IsStr. This function returns the function of y which makes up the body of IsStr. The arguments of IsStr become "own" variables for the predicate function.

provide efficient implementations. For example, testing for nil tuples might speed up the function but it would only complicate the program unnecessarily. For this reason many of the function definitions will not be optimal. Any implementation could of course recognize these special cues and simplify the resulting functions.

The Constructor Building Function MakeStr

It is natural to assume the argument to the constructor will be a tuple of components. In this case this tuple can be used as the tuple which represents the data. The selector decoding consists of finding the index of the selector in a tuple of selectors in the proper order. The selected component is generated by applying the argument tuple to this index. The special selectors (tag, domain, constructor) are handled by tests for their occurrance before the selector is decoded. The decoding procedure and constructor builder are given in figure 8.

Since the MakeStr function produces a function producing function, it is easier to see how it works from a picture of the environment of each function. In figure 9 there is a representation of what happens when MakeStr is applied to an argument tuple consisting of a tag (argl) and a selector set (arg2).

```
def Decode (y,Sel) = D (Order Sel)
      where rec D k =
         k eq 0 -> 0 | y eq (Sel k) -> k | D (k-1)


def MakeStr (Tag,Sel) = Constructor
      where rec Constructor (Tuple) =
         fn y. y eq tag -> Tag
               | y eq domain -> Sel
               | y eq constructor -> Constructor
               | ( let k = Decode (y,Sel) in
                    k eq 0 -> undef | Tuple k ) }
```

figure IV.8 The representation of MakeStr. This is a function producing function which produces the function Constructor. When Constructor is applied to a tuple it returns the function of y which represents the data. The tag and selectors are "own" to the function Constructor and these plus the data tuple are "own" to the data function. The result of the data function is the special atom <u>undef</u> if the selector is not defined.

The constructor is created in step 1) and this is used in step 2) to build an instance of a data function

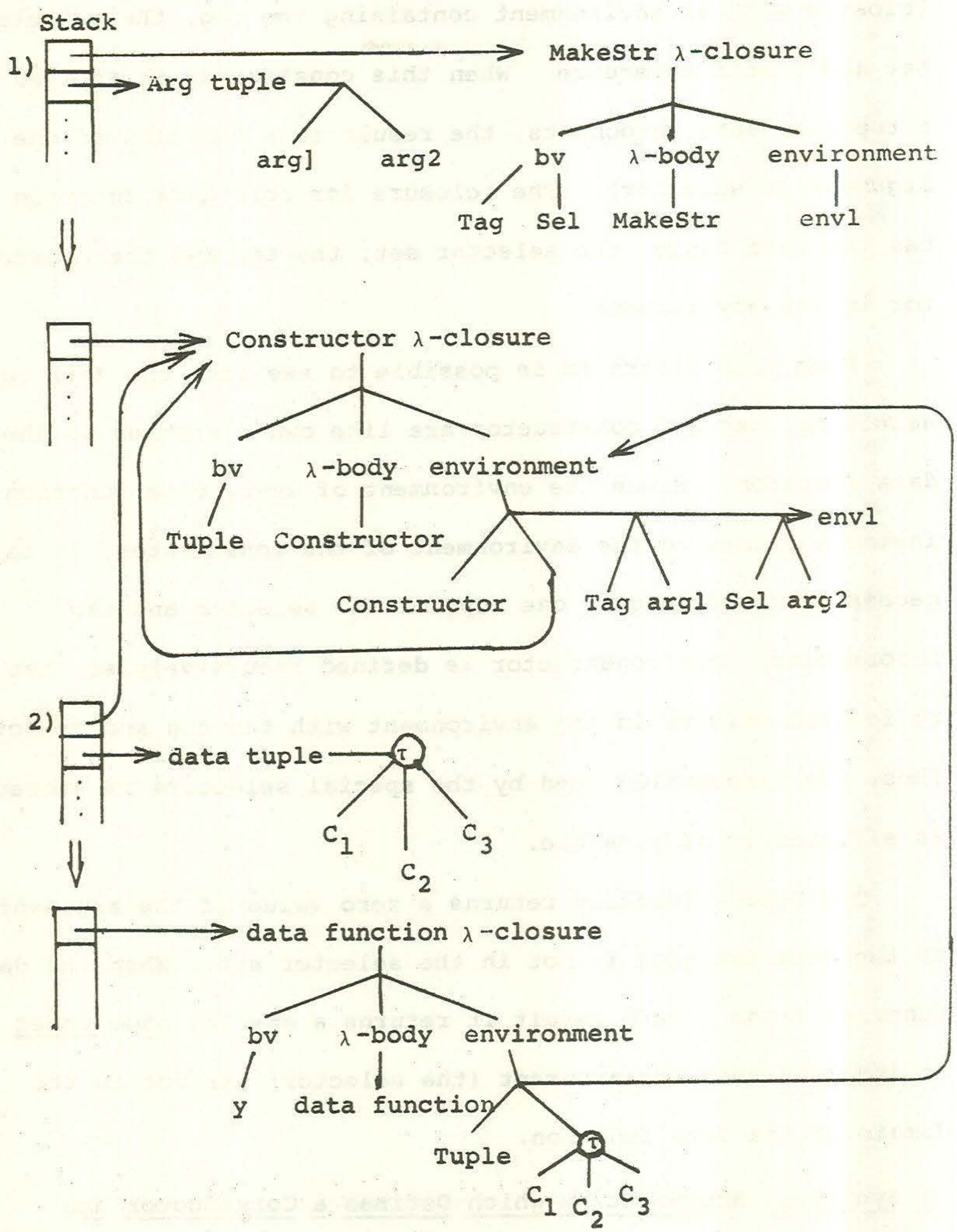

figure IV.9 The environment of the constructor and data function

The result of this application is the constructor which is a λclosure with an environment containing the tag, the selector set and a self reference. When this constructor is applied to a tuple of data components, the result is a function of one argument (a selector). The λclosure for this data function has the data tuple, the selector set, the tag and the constructor in its environment.

From this figure it is possible to see that the data tuple, selectors, tag and constructor are like own variables to the data function. Since the environment of every data function instance points to the environment of the constructor, it is necessary to store only one copy of the selector and tag information. The constructor is defined recursively so that it is also defined in the environment with the tag and selectors. Thus, the information used by the special selectors is stored as efficiently as possible.

The Decode function returns a zero value if the argument to the data function is not in the selector set. When the data function finds a zero result it returns a special atom <u>undef</u> to indicate that its argument (the selector) was not in the domain of the data function.

A Syntactic Abbreviation which Defines a Constructor and Complex Predicate

So far the syntax defined allows simple data structures to be constructed. We begin extending this facility by introducing an abbreviation for a special case of the predicate definition. Consider the definition

def LIST which is

(6) (HEAD also TAIL)

else IsNIL

In this definition there are two alternatives exactly one of which is a constructed object. However, the constructed object is without a name of its own and is indicated only by the <anonymous-structure> HEAD also TAIL. If a list of alternatives for a predicate definition includes exactly one constructed object, the name (tag) for that constructed object may be elided. In such a case the tag of the constructed object will be taken from the predicate name. For the above definition (6) the tag of the constructed object will be "LIST" and the constructor for it will be "MakeLIST."

Syntactically, this is facilitated by changing the syntactic rule for <named predicate> as given in (5) and by adding another rule

<named predicate>::=ATOM which <anonymous predicate>
| ATOM which <structured predicate>

<structured predicate>::=is(<anonymous structure>)
$\{$else <predicate designator>$\}_1^\infty$

or, in abbreviated form

P::=ATOM <u>which</u> P1 | ATOM <u>which</u> P2

P2::=<u>is</u>(S1) {<u>else</u> P3}$_1^\infty$

Since a <named predicate> with an <anonymous structure> as the sole alternative would be exactly equivalent to a <named structure>, it appeared to be less confusing if additional predicate alternatives were required. Therefore, at least one <predicate designator> must appear after the <anonymous structure>. The new abstract syntax is given in figure 10. Note that the node tag for "ATOM <u>which</u> P2" has been changed to "is/has" to make it possible to distinguish the two forms of P when they are encountered. This information is used in the standardization process.

The abstract syntax tree for definition (6) is given in figure 11. Note that <u>else</u> is used to designate both P1 and P2 and only by examining the form of the syntactic variable preceeding the first <u>else</u> are they distinguished. This double use of else is possible because the two forms of P are distinguished. The processed form given in figure 12 defines only a predicate and a constructor but the predicate builder has a one tuple with the single predicate IsNIL as an argument.

figure IV.10 Abstract syntax for abbreviated predicate-constructor

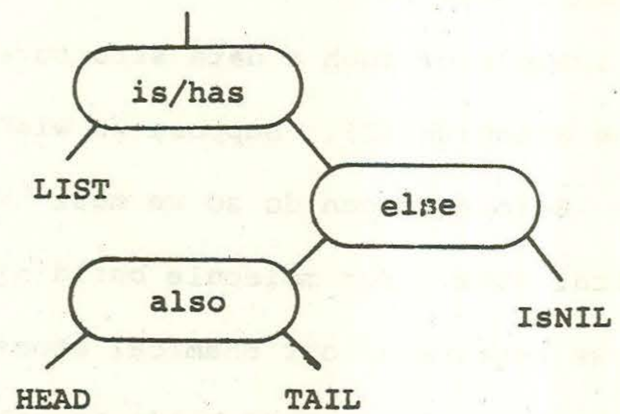

figure IV.11 Abstract syntax tree for definition (6)

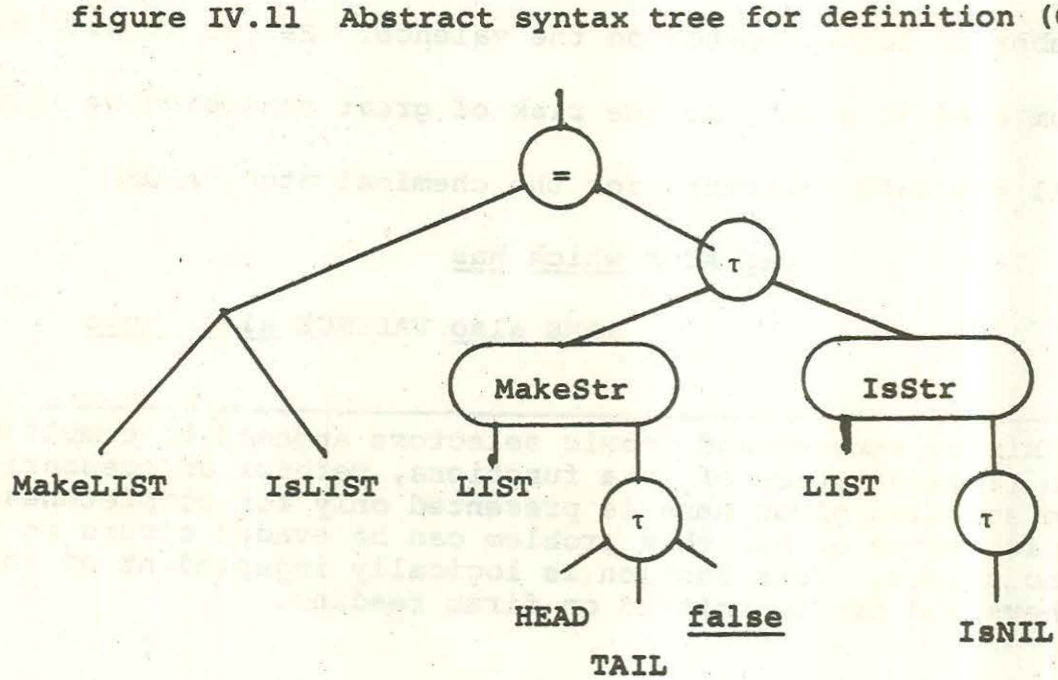

figure IV.12 The standardization tree for definition (6)

Data Functions with a Mixed Domain*

A careful reader may have noticed that in the standardized form of the above defunctions an extra selector, false, was always appended to the selector set argument of MakeStr. This argument is needed to allow for data functions defined with both atomic and integer selectors. These are refered to as mixed domain data functions.

As an example of such a data structure we borrow an example from Standish[33]. Suppose we wish to represent a molecule. Before we can do so we must have a representation for a chemical atom. For molecule building there are three properties we require of our chemical atoms. They must have a name, a valence and a set of bonds to other atoms. The number of bonds depends on the valence. Hence, it will vary from atom to atom. At the risk of great confusion we will call the data structure for the chemical atom, ATOM

def ATOM which has

(8) NAME also VALENCE also tuple

---

* Mixing integer and atomic selectors appears to complicate the representation of data functions, perhaps unnecessarily. The solution given here is presented only for completeness; a discussion of how this problem can be evaded occurs in the conclusions. This section is logically independent of the others and can be omitted on first reading.

We have used the keyword <u>tuple</u> to indicate that an indefinite number of integer indexed components will be part of an instance of the data function for atoms.

<u>The Syntax for Mixed Domain Selectors</u>

The syntax for mixed domains is created by modifying the <anonymous structure> syntax (3).

$$
\begin{aligned}
\langle\text{anonymous-structure}\rangle ::= & \{\langle\text{selector}\rangle\ \underline{\text{also}}\}_1^{\infty}\langle\text{selector}\rangle \\
& \mid \{\langle\text{selector}\rangle\ \underline{\text{also}}\}_0^{\infty}\ \underline{\text{tuple}} \\
& \mid \underline{\text{only}}\ \langle\text{selector}\rangle
\end{aligned}
$$

or in abbreviation

$$S1 ::= \{S2\ \underline{\text{also}}\}_1^{\infty} S2\ \mid \{S2\ \underline{\text{also}}\}_0^{\infty}\ \underline{\text{tuple}}\ \mid\ \underline{\text{only}}\ S2$$

The reader may notice that a <u>tuple</u> may occur without any symbolic selector being specified. Because it is a reserved word no syntactic ambiguity occurs in this case. The abstract syntax for the modified rule is given graphically in figure 13. Figures 14 and 15 give the abstract syntax tree and the standardized tree for definition (8).

<u>The Reason Why the Integer Selectors Follow the Atomic Selectors</u>

As we noted above different chemical atoms have different numbers of bonds depending on the valence. Therefore, it must be possible to construct data structures where the extent of the tuple part is variable. As the use of <u>tuple</u> suggests, the tuple part is variable in extent and the integer selectors associated with

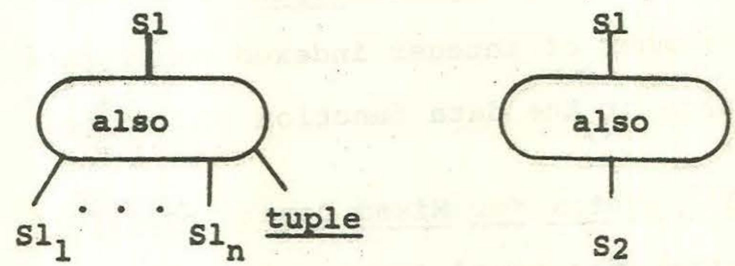

figure IV.13 Abstract syntax for mixed and singular definitions

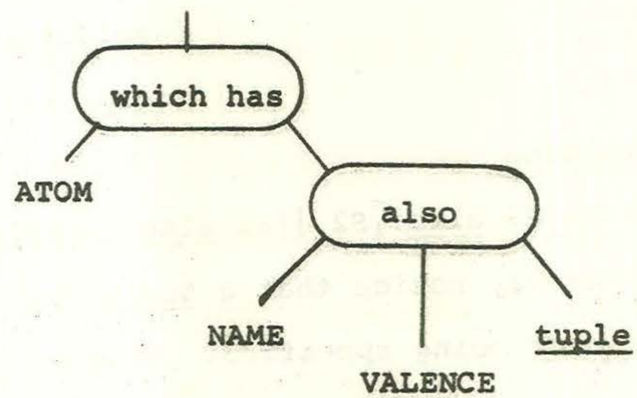

figure IV.14 Abstract syntax tree for definition (8)

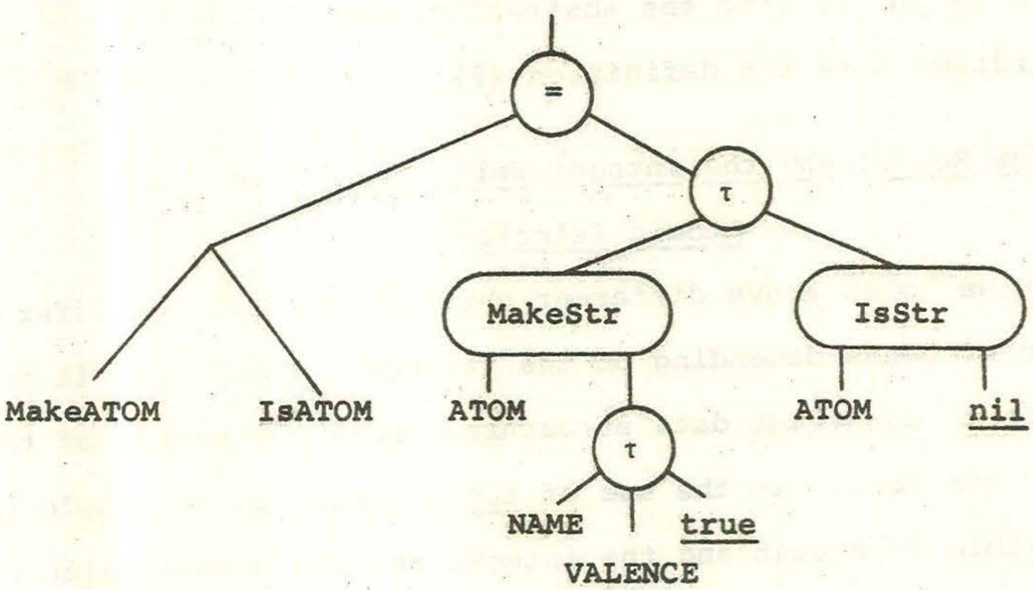

figure IV.15 Standardized tree for definition (8) any instance of a mixed domain data function range from 1 to the order of the tuple part. Of course, the tuple part may also be nil.

It is no accident that the tuple part of a structure follows the symbolically selected parts. The components of the tuple part are included in the tuple of data on which the structure is defined. Since the number of tuple components may vary, it is necessary to have a way of identifying the tuple part components. There are always a fixed number of components with atomic selectors and these must always be present. Therefore, the simplest way to identify the tuple components is to put them after the ordered set of symbolically selected components. Then the length of the argument to the constructor defines the length of the tuple part.

For example, consider the construction of a typical chemical atom, say carbon. The carbon atom has a valence of 4 so four bonds, represented by pointers, are required. A typical construction using definition (8) might be

```
MakeATOM('CARBON',4,ptr1,ptr2,ptr3,ptr4)
```

where ptri represents a link to another chemical atom. The bond pointers would be selected by 1,2,3 and 4.

Putting the tuple last is convenient for a second reason. It makes it possible to augment structures just as tuples are augmented. For example, it might be desirable to construct only the symbolic part of an atom initially and to add the bonds later. This can be done by defining a function which first extracts the components of the existing structure and puts them into a tuple. The new component is added using aug and finally a new copy of the structure is constructed (see figure 17). Since the tuple part is last, the added component becomes the last component of the tuple part.

An Alternative Mixed Domain Definition

The S-PAL approach is certainly not the only way of defining a mixed domain data structure. Another alternative is given by Standish [33]. He chose to allow the user to refer to a component either by its selector name or by the ordinal for its position in the data structure definition. Therefore, a component might have two selectors. If a user wanted only the integer selector, the component was defined by a place holder and the selector name was omitted. The place holder he used was the type specification for the component. He did not, however, allow for augmenting a data structure. Instead he provided families of data structures where each structure had a different number of components.

The main advantage to a non-augmentable set of selectors is that it is possible to specify distinct type information for each component in the definition. When an indefinite number of components may occur it is only possible to specify a type which every component must have. This question will be treated in more detail in chapter V. It would be possible to include a fixed set of integer selectors in S-PAL data functions by allowing integers as well as atoms as <selector>s. However, this point will not be persued.

The Extension of MakeStr to Allow Mixed Domains

It is now possible to interpret the truth value which was appended to the selector set. If this value is false, then the data function is of fixed sized with atomic selectors. If the value is true, then the data function has a variable tuple part and different instances may have different size.

Unfortunately, this simple extension makes the function MakeStr much more complex. It is now necessary to have the data function representation recognize two different types of selectors. The atomic selectors are still looked up in the selector list while the tuple representing the data is applied to the integer selectors directly. The variable components are selected by adding the length of the fixed (atomic) part to the selector value. Since only the atomic selectors are stored in the constructor, it is necessary to build the full selector sets when the special selector domain is given. This is done in the auxilliary function Buildset. The new form of MakeStr is given in figure 16.

An aug-like Operator for Data Functions

Because the mixed domain data functions may grow in size it should be possible to write a function which will augment the tuple part. This function will produce a new augmented data function just as aug produces a new tuple. This is necessary to avoid side effects when the original structure is used. That is, augmenting the structure should not affect other uses of that structure.

The function AuG in figure 17 uses two of the special selectors. It first decomposes the current data function into its components using the auxiliary function Destroy. The tuple of components is then augmented and the constructor is applied to the augmented tuple to give the augmented data function. The function Destroy is implemented as a primitive in some languages such as POP-2 [4]. It is called a destructor and produces the tuple which was originally used to construct the function.

Structures with Explicitly Enclosed Substructures

The structure facilities defined so far provide for constructing one level structures. If a multilevel structure such

```
def  MakeStr (Tag,Sel) =
        let n = Order Sel - 2 in Constructor
           where rec Constructor (t) =
              [ fn y. IsATOM y ->
                    y eq tag -> Tag
                  | y eq domain -> Buildset (Sel,t)
                  | y eq constructor -> Constructor
                  | { let k = Decode(y,Sel) in
                         k eq 0 -> undef | t k}

                Sel (Order Sel) -> t (n+y) | undef ]

def  Buildset (Sel,t) = R (Order t - Order Sel + 1)
        where rec [ R k = k eq 0 -> Q (Order Sel - 1)
                                  | Aug (R (k-1)) k
                  and Q m = m eq 0 -> nil
                                  | Aug (Q (m-1)) (Sel m)]
```

figure IV.16 MakeStr function for mixed domains. This MakeStr is almost identical to the previous one except for the non-atomic selectors which are used to select the tuple part. Buildset has two recursive searches. Q builds the tuple of atomic selectors and this is augmented by R to include the integer selectors.

```
def  AuG (Structure,Object) =
        Istuple Structure -> Aug Structure Object
          | f(Aug [Destroy Structure] Object)
              where f = Structure constructor

def  Destroy (Structure) =
        let A = Structure domain in Q (Order A)
           where rec Q k = k eq 0 -> nil
                                  | Aug [Q(k-1)] [Structure(A k)]
```

figure IV.17 A function for augmenting data functions

As a binary tree is desired, it is necessary to use a multistep construction. This can be done by first constructing all lower levels of the structure and then constructing the next higher level using the previously constructed structures as arguments to the constructor for the higher level. Alternatively, the construction can be done from the top down using loc's and updating pointers to the lower levels when they are constructed. In general, there are no bounds on the size or complexity of such a structure. It can grow dynamically at run time. It is also impossible to predict the storage requirements for such a structure at compile (translate) time.

A Different Approach

This section presents an alternative method for defining multilevel structures. This technique, which might be called static structuring, is useful when the substructures have a fixed relationship to the major structure and this relationship is known at compile time. If each substructure has a well defined position in the structure, is of a known type, and is always present, then it is possible to predict the storage requirements for the structure with its substructures. It is also possible to use such techniques as contiguous storage to reduce the need for pointers within the structure. This in turn makes references to parts of the structure simpler. When the user provides the static or fixed structuring information, the compiler can use this to optimize resource usage for that structure type.

If the main idea of static structures is that all of the information should occur together, why not represent it by a single structure with many components? This certainly could be done but it would inhibit one of the main uses for data structures. One of the reasons for grouping data into a structure is that the components all have some relation to each other which the user finds convenient to make explicit by grouping them and giving the grouping a name. If he is forced to use a large structure with all the components at the same level, he is unable to group subsets of these components. This grouping of subsets is important because he may wish to specify operations on subsets without having to list all the members of the subset.

Reducing Naming Conflicts

When a large data structure such as a binery tree is created dynamically there is no problem in refering to a subpart of the structure. If the anchor node of the structure is known, then any substructure can be accessed by applying the data function for the anchor node to a string of selectors which indicate a path to the desired substructure. Therefore, any such substructure is referenced by an anchor data function and string of selectors. This is a computed reference.

In the case of statically defined substructures, it is also possible to use a computed reference to access substructures. However, as we shall see when some additional properties of static structures are presented, it is convenient to have a name for the static substructures. Since the names of structures are derived from the tag, this creates a problem of possible name conflicts in structures with similar substructures. Two substructures may have the same tag because the data they contain is related. For example, one such substructure may contain a subset of the information contained in another. However, it must be clear which is intended in any particular use because their actual structure may differ.

This problem can be solved by qualifying the name of the substructure with the names of all the structures and substructures in which it is embedded. This produces a tag or name which identifies the substructure as belonging to a particular place in a particular structure. This name is formed by concatenating all the names in the path to the substructure. This is analogous to a compile time evaluation of a computed reference to that substructure. However, the qualified name, since it is defined at compile time, may be used as a bound variable which is not possible with computed references. This becomes important in defining constructors and predicates.

An Example of a Static Structure and Its Use

The utility of static substructures may become more apparent from a simple example. Consider a typical payroll file which might have a structure or record consisting of an identification substructure, an address substructure, a salary substructure and a year to date substructure. Each of these substructures has a different function, but is always present for every employee.

Now consider a typical weekly update operation. A set of time records which have the time worked by each employee will be used to find the employee's record, update it, and produce a paycheck record. The time record will typically have an identification substructure and an hours worked substructure. The identification substructure would probably contain a subset of the payroll identification substructure. For example, it might contain an id number and a name while the payroll identification substructure might also have the social security number.

In practice, the identification substructures would be used to find the payroll record corresponding to the update record. For example, the search program would compare the id number of the identification substructure in the current payroll structure with the id number if the identification substructure of the update structure. Since the element names are identical they are only distinguishable by the structure in which they occur. Thus, a comparison can only be written with the qualified names.

A second point to notice is that all the substructures of the payroll record are used when the structure is updated. The identification substructure is used to find the correct record. The salary substructure is used to compute the amount to be paid. The year to date field is updated and copied into the paycheck information and the address is used to mail the paycheck. The substructure groupings correspond to different operations performed on the payroll record, but they are all accessed in the update process.

Optimizing such data accesses is particularly important on computer systems with multilevel stores (e.g., paging systems). In this case there is usually a large cost associated with a reference to data which is not currently in the top level store. Therefore, techniques, such as contiguous storage, which keep a structure with frequently accessed substructures on a single page increase the operating efficiency of the programs which process the structure.

Properties which Make Static and Dynamic Substructures Compatible

Before presenting the syntax for substructure definitions it is necessary to discuss several properties a substructure facility should have. First, from an operational point of view it should not matter whether a substructure was declared statically or was dynamically inserted at run time. For example, a subroutine should not be able to distinguish whether an argument is a static or dynamic substructure or even if it is a substructure at all. In either case, the argument should appear to be a structured Rvalue. This makes it possible to use the static substructures as if they were defined independently as major structures. That is not embedded in another structure.

If substructures are to be truly independent of the major structure in which they are embedded it must also be possible to construct the substructures and use the results of these constructions to build the major structure. For example, it should be possible to build an hours substructure and an identification substructure and combind these into a time structure. This capability is needed when various components of a structure are computed or constructed in different subroutines.

This is made possible by defining a subconstructor for each substructure in the structure definition. The name of this subconstructor is qualified by the name of the major structure and substructures in which it is embedded. This makes it possible to specify which of several substructures with the same simple name is to constructed. This is also one reason why the name of a constructor is derived from the structure definition rather than letting the user bind his own name to the constructor.

It is also possible to construct a major structure and all its substructures in a single operation. The argument to the constructor is still a tuple of components but when a component corresponds to a substructure in the major structure, that component can be a tuple of components for the substructure. The components of the substructure might also be tuples of components for lower level substructures. If the constructor for a structure finds a tuple of components where it expects a substructure, then the constructor for that substructure is used to build a constructed object from the tuple. This process is recursive, hence it may occur to any depth. With this definition it is possible to mix previously constructed substructures with implicitly constructed ones.

<u>An S-PAL Definition of the Payroll Update Structure</u>

The following is one possible definition of the update structure mentioned above

```
def  TIME which has
        (ID which has
(10)                ID_NO also NAME)
        also
        (HOURS which has
                    WORKED also SICK also VAC)
```

This definition defines a major structure which will be tagged by TIME and two substructures with tags TIME.ID and TIME.HOURS. As we noted above, the qualification of the substructure tags makes it possible to distinguish substructures with the same unqualified name. This property is used in the predicates for these substructures which will only yield true for a structure with the correct fully qualified tag.

However, the unqualified name of a substructure is used for the selector name of the component of the major structure corresponding to the substructure. For example, the ID_NO component is accessed by TIME ID ID_NO. This first produces the substructure TIME.ID from which the component ID_NO is selected.

An Alternative Notation for Functional Application

Sometimes it is more convenient to list the selectors in reverse order to indicate you want the ID_NO component of the ID component of TIME. Therefore, an alternate notation for functional application is provided. This expands the current notation for functional application and has lower precedence.

R::= R2 <u>of</u> R | R1

R1::= R1 R2 | R2

(11) R2::= NUMERIC | QUOTATION | TRUTHVALUE

| NAME | <u>nil</u> | (E) | [E] | ATOM

For example, "ID_NO <u>of</u> ID <u>of</u> TIME" is equivalent to "ID_NO <u>of</u> TIME ID" which is equivalent to "TIME ID ID_NO." However, parentheses are needed to say "(ID <u>of</u> TIME) ID_NO".

The Syntax for Static Substructures

The syntactic extension for substructures is trivial. All of the actual work is done in the standardizing routines. We also include here the change which allows subpredicates.

<selector>::= ATOM | (<named structure>)

<predicate designator>::=<rand> | <u>is</u>(<named structure>)

| <u>is</u>(<named predicate>)

(12) or in abbreviated form

S2::= ATOM | (S)

P3::= R2 | <u>is</u>(S) | <u>is</u>(P)

The interpretation of the expanded <selector> is that the <named structure> is defined as a substructure and the name becomes the selector for that component. The tag of the substructure and the name of its constructor are qualified by the names of all statically containing structures.

The interpretation of the subpredicate <named predicate> is much simpler. It defines an additional disjunctive predicate. Basically it allows a subset of the alternatives in a disjunctive predicate to be given a name of their own. There is no constructor or tag to be concerned with, so the name of the subpredicate is not qualified by the predicate name. The complete abstract syntax for the structure definitions is given in figure 18.

The abstract syntax tree for definition (10) given in figure 19 is not too much more complex than that given for the previous definition. Basically it shows the nesting relationship of the substructures. It is the standardized version of definition (10) which shows the additional complexity of substructures. This tree, given in figure 20, is in the form of a complex simultaneous definition similar to that which occurs in the standardization of definitions connected by <u>and</u>. Thus, the constructors and predicates for the structure and all contained substructures are defined simultaneously.

This is not the only alternative. It would also be possible to define the structure in a context where the substructures were already defined. From the viewpoint of simplicity of specification of the standardization process, this is not a good choice because it means inverting the tree structure and

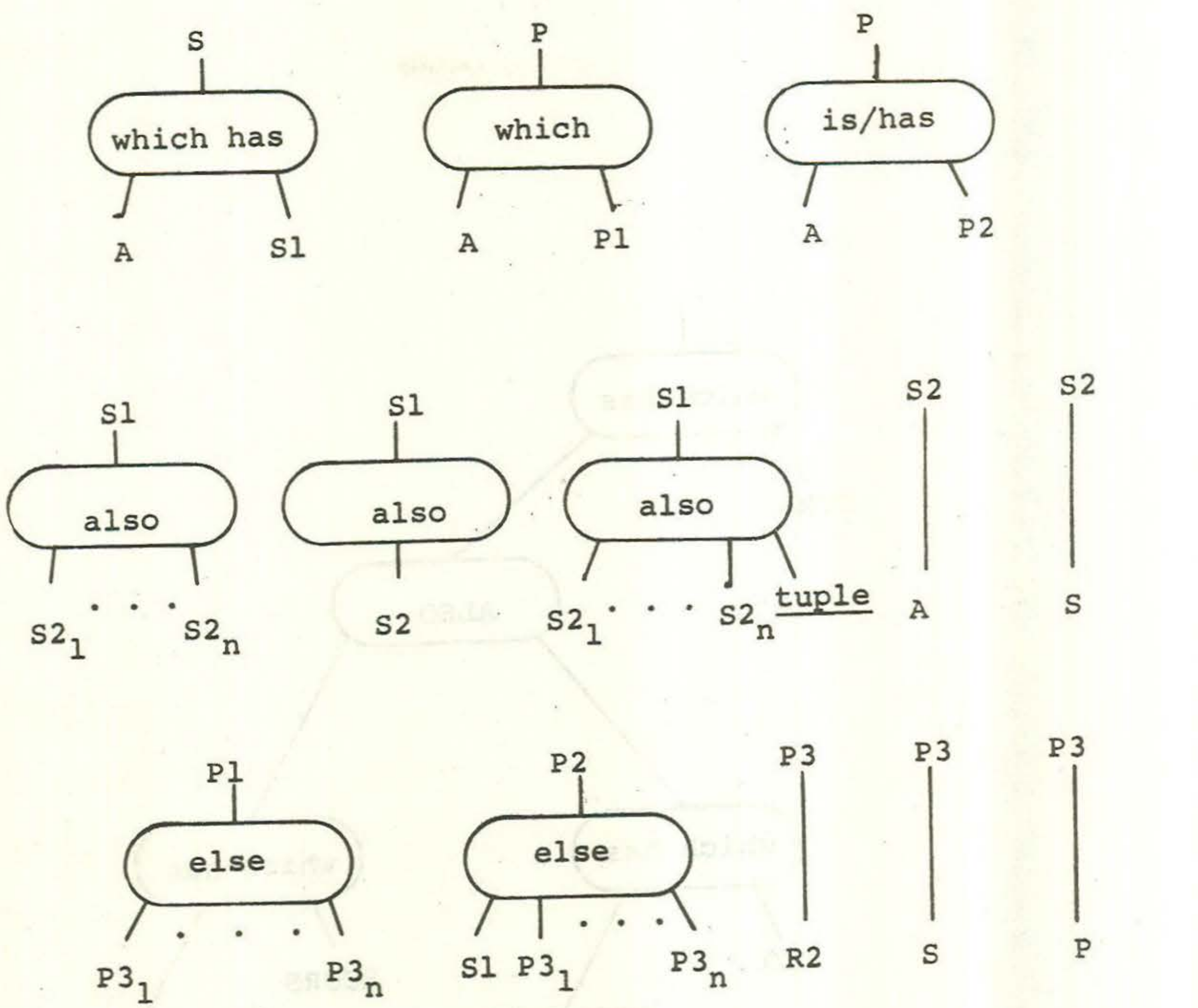

figure IV.18 The complete abstract syntax for structure definitions. The syntactic categories which are not defined by nodes are

A which represents an atom

and R2 which represents a function or more explicitly a predicate.

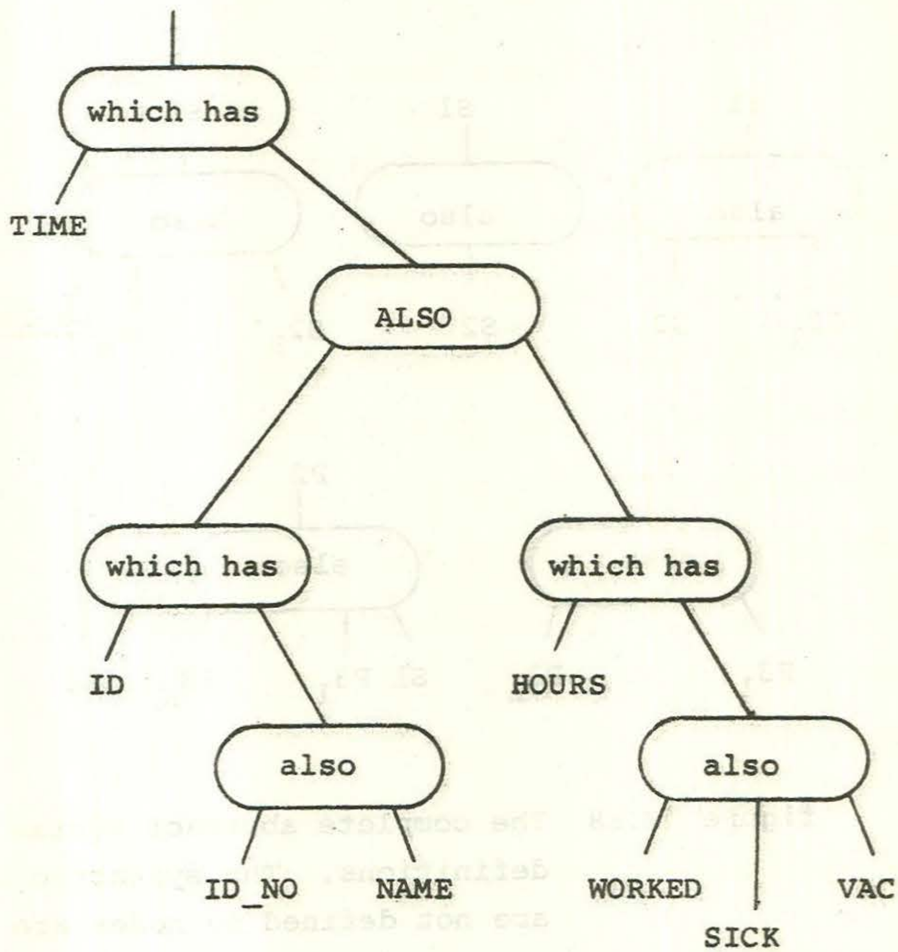

figure IV.19 The abstract syntax for definition (10)

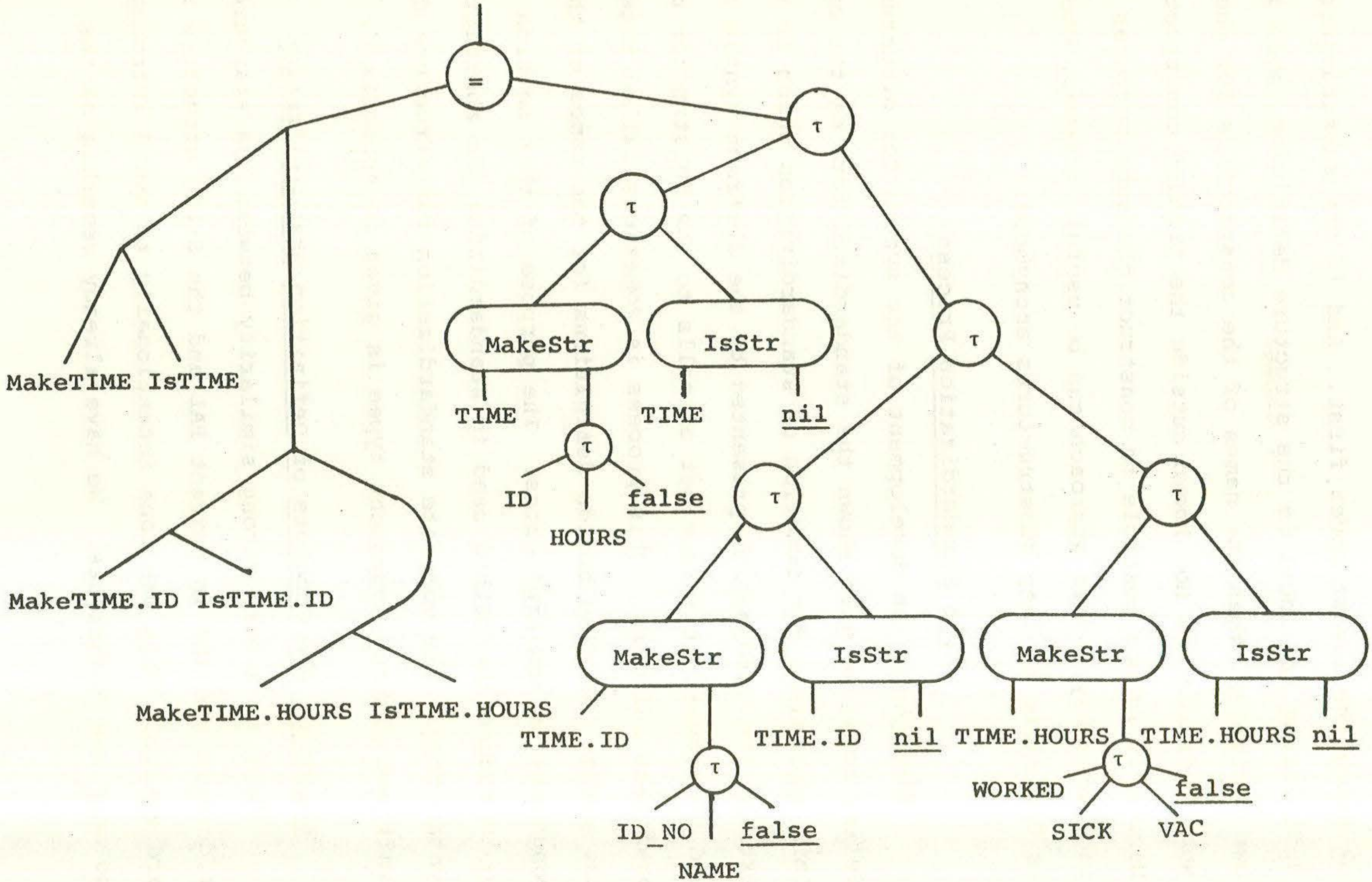

figure IV.20 The standardized form of definition (10) defining the lowest nodes first. And if the substructure definitions are local to the structure definition, much like own variables, then the names of the constructors for these substructures are not known outside the primary constructor. Hence, it is not possible to construct the substructures independently. This approach can be useful, however, when it is desirable to keep substructures anonymous.

The Standardization Process

Throughout the development of the syntax for structure definitions, we have shown the standardized form of the examples. This emphasises the function of standardization which is to extract the information presented by the abstract syntax tree and to convert it into a set of calls to the constructor and predicate builders. This process is sometimes called interpreting the parse. It also builds definitions for the names of the constructors and predicates. The purpose of this section is to introduce the method used in standardizing the abstract syntax tree. The complete standardization for structure definitions without component types is given in Appendix C.

Extending the Concepts of Definition Standardization

There is a very strong similarity between the standardization of definitions (D) in current PAL and the S-PAL structure standardization. This was done intentionally to avoid introducing too many new techniques. We have already remarked on the similarity between substructures and simultaneous definitions. This will be developed in greater detail below.

The structure standardization is added to the definition standardizer D. There are three new alternatives, a structure definition (NS), a predicate definition (NP), and a combined or abbreviated predicate and structure definition (NB). These routines are applied to standardized versions of the components of the corresponding abstract syntactic node just as AD is applied to standardized versions of the components of the "<u>and</u>" node. In fact, the routines NS, NP and NB are used recursively for substructures the same way D is used for subdefinitions.

A Pictoral Representation of the Standardizing Process

To help explain the action of the standardizing routines, we will use a pictoral representation of the transformations being performed. These only indicate the steps of the standardization process and do not always correspond exactly to the operation of the standardizing functions. The major difference is that some structures shown as a single object are actually handled as separate components in the functions since it was simpler to remember implicitly the connections between the parts.

Standardizing Definitions without Sublevels

The major portion of the standardizer is needed to handle substructures. A simple structure such as definition (1) is converted to standard from relatively directly. The first step is to collect the set of selectors into a tuple. This is performed by the unnamed structure processing function (US). Figure 21 shows the result of US for a set of atomic selectors. If the tuple option was present the final component would be <u>true</u>.

The next step is to build a simple definition for the constructor and predicate. This is separated into the two steps shown in figure 22. The first step is to create the names of the two functions from the tag of the structure. This is done using the metafunction "QualN" which concatenates the string which is its first argument with the string or atom which is its second argument. If the second argument is an atom it is converted to the printable representation of the atom before concatenating.

The second step is performed in the function simpleNS and consists of building argument lists for MakeStr and IsStr. The function SimpleNS actually constructs combinations wherein MakeStr and IsStr are applied to their arguments, but to save space this is represented by the nodes MakeStr and IsStr in the pictoral form.

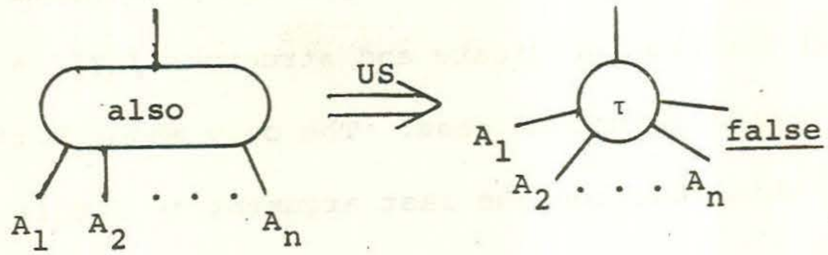

figure IV.21 Simple selector processing. The final component is false to indicate the tuple part was not present in the abstract syntax tree.

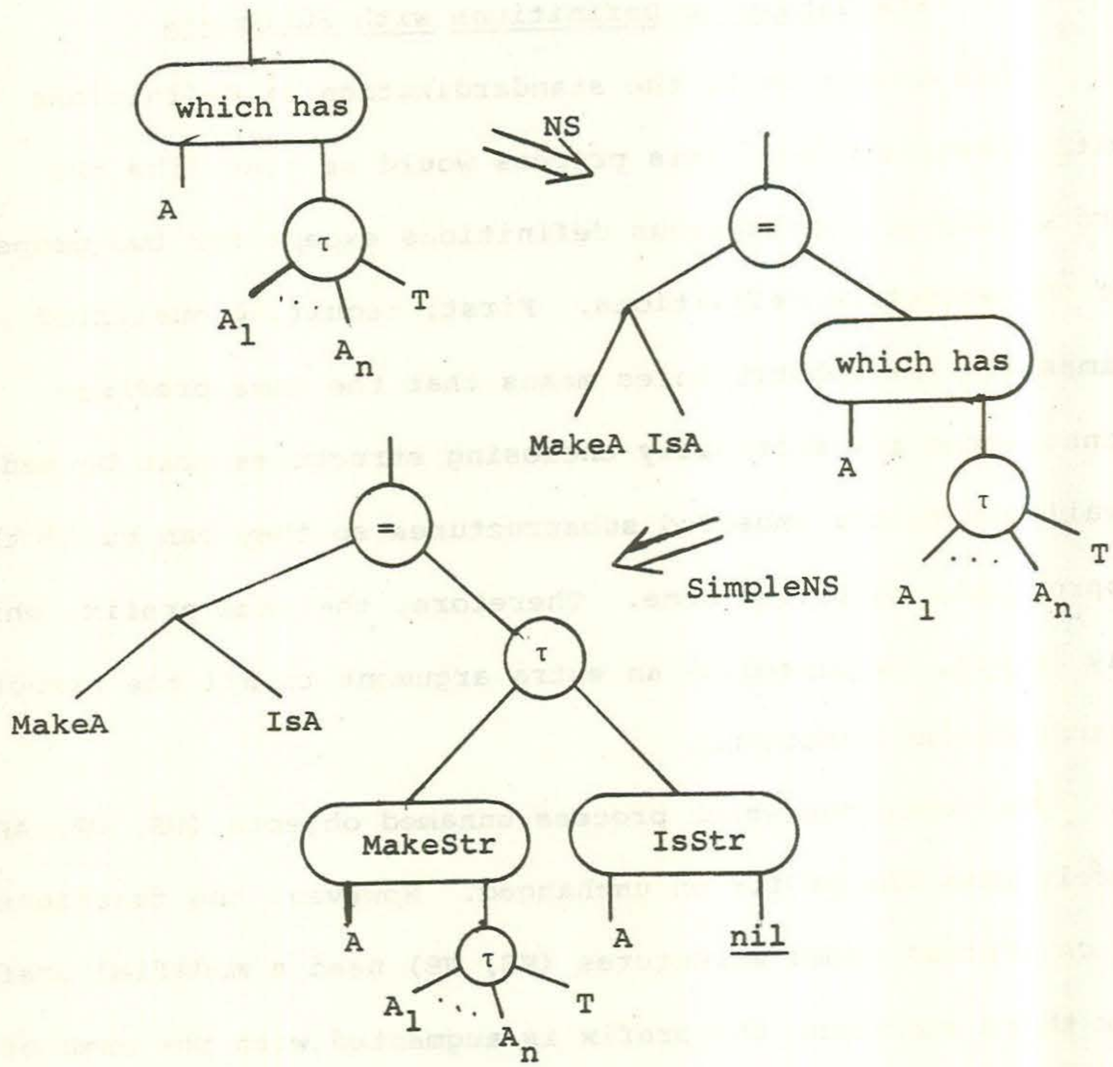

figure IV.22 Standardization of name structures without substructures

The processing for the simple forms of named predicates (NP) and combined predicate and structure (NB) is very similar to the simple structure case. The only major difference is in NB. In this function the last argument to Simple NS, the list of alternative predicates, is not nil but consists of the predicates from the <structured predicate>(P2).

Standardizing Definitions with Sublevels

This brings us to the standardization of definitions with substructures. This process would be just like the processing of simultaneous definitions except for two properties of the structure definitions. First, requiring qualified names for the substructures means that the name prefixes constructed for statically enclosing structures must be made available to the embedded substructures so they can build the appropriate qualified name. Therefore, the name prefix, which may be nil, is passed as an extra argument to all the structure standardizing functions.

The functions which process unnamed objects (US, UP, AP) merely pass the prefix on unchanged. However, the functions which process named structures (NS, NB) need a modified prefix. For these functions the prefix is augmented with the name of the structure (substructure) being processed. Note that the named predicate processing function (NP) does not require a qualified name, so the prefix is not changed. Actually two names are provided to the routines for named objects (NS, NP, NB). The first is the unqualified name which is used in the context of selection and the second is the qualified name.

The second reason why processing of substructures differs from simultaneous definitions is that information collection is being done concurrently with the definition of the constructors and predicates for the substructures. That is, the unqualified names of the substructures also serve as selector names for the components of the enclosing structure. Therefore, it is convenient to build two tuples of information for substructures and subpredicates. The first tuple consists of the selector set and the second tuple consists of the tree of simultaneous definitions of substructures below the anonymous structure currently being processed.

Processing Each Component Definition

Since a substructure has the same syntax as a major structure the processing function Sub is introduced to mimic that part of D which deals solely with structure or predicate definitions. Since Sub may be invoked from either a predicate or a structure definition and in general different information is needed, its result is a 3-tuple. The first component is the unqualified name of the substructure or subpredicate processed by Sub.

This is used for the selector name. The second component is the name of the predicate for the substructure or subpredicate. It is used in constructing predicates and will be used in the type system introduced in Chapter V. The final component is the simultaneous definition for the substructures below the current one. This process is represented pictorially in figure 23. Only a single level is shown because of space considerations. The label "Subs" is introduced to give a name to the 3-tuple. Note that the prefix for name qualification is used.

Combining Individual Component Definitions

Using the analogy with the standardizing of simultaneous definitions the next step would be to combine the definitions of all the substructured components of the current substructure into a single simultaneous definition. However, this must be done in two steps because the component definitions must be separated from the other information produced by Sub.

Since we want to collect both the selector set and the set of all structures defined at lower levels, two tuples of information are constructed. The function Split is used to call Sub for each component and to put the resulting information in the correct tuple. Since Sub always returns a 3-tuple, the selector for the current component is obtained from the first

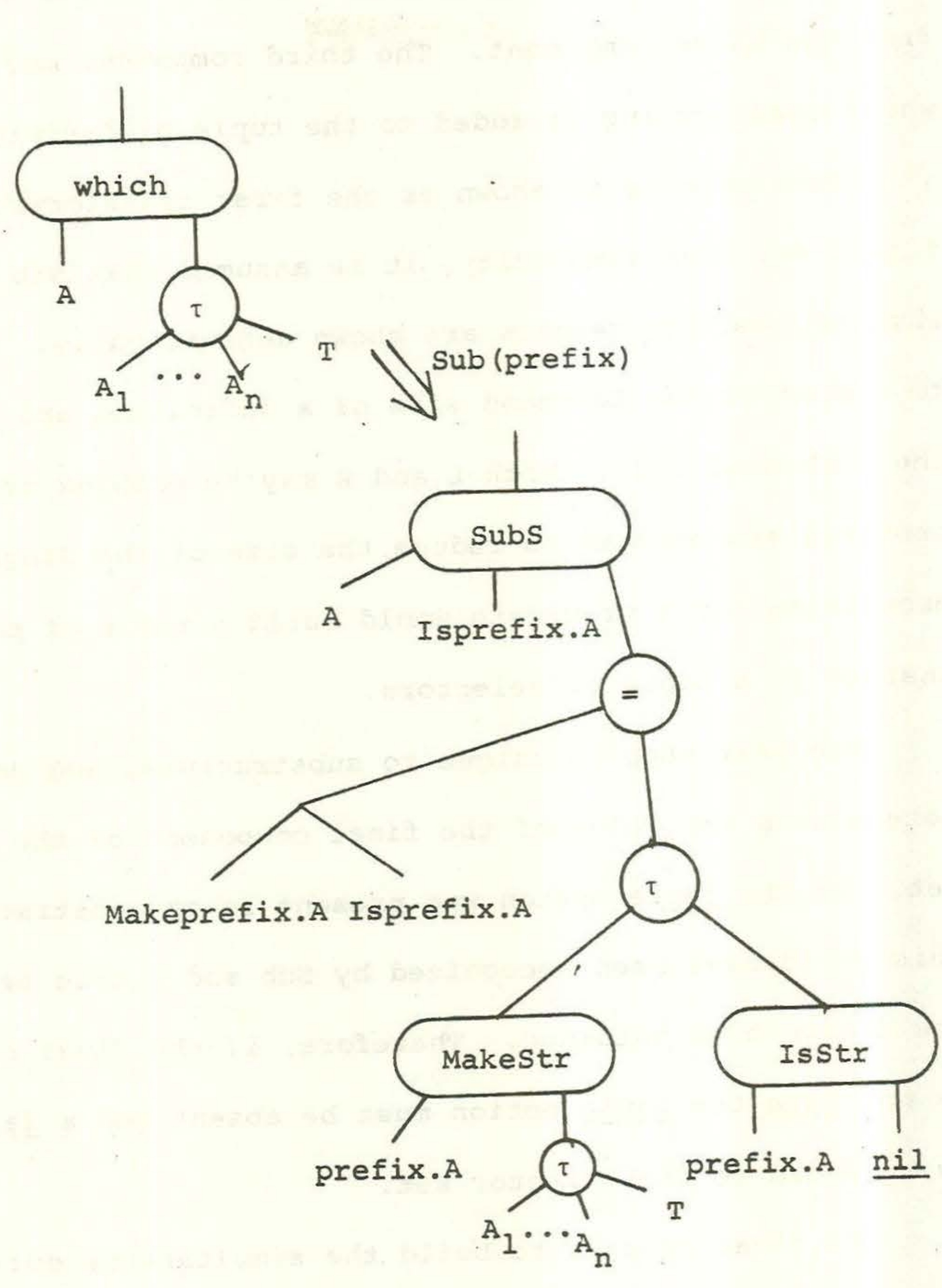

figure IV.23 The result of processing substructures. The transformation is performed by Sub and uses the name prefix which is indicated as an argument to Sub. This prefix is used to qualify the constructor and predicate names as shown.

component of the Sub result and the lower definitions, if any, from the third component. The third component may be nil in which case nothing is added to the tuple of definitions.

This process is shown as the first transformation in figure 24. For simplicity, it is assumed that Sub was already invoked and the results are shown schematically. L is used to represent the lefthand side of a definition and R represents the righthand side. Both L and R may be complex trees. Also prefixes are omitted to reduce the size of the diagram. The processing for a predicate would build a tuple of predicates instead of a tuple of selectors.

The next step is unique to substructures and consists of determining the value of the final component of the selector set. If the tuple option was present in the abstract syntax this would have been recognized by Sub and a true selector would have been returned. Therefore, if the final component is not true the tuple option must be absent and a false value is appended to the selector set.

The final step is to build the simultaneous definition for the subobjects and to combine the selector set (predicate set) with the simultaneous definitions. This is performed by the routine Combine. It first checks to see if any substructures (subpredicates) were defined at a lower level and if not, it

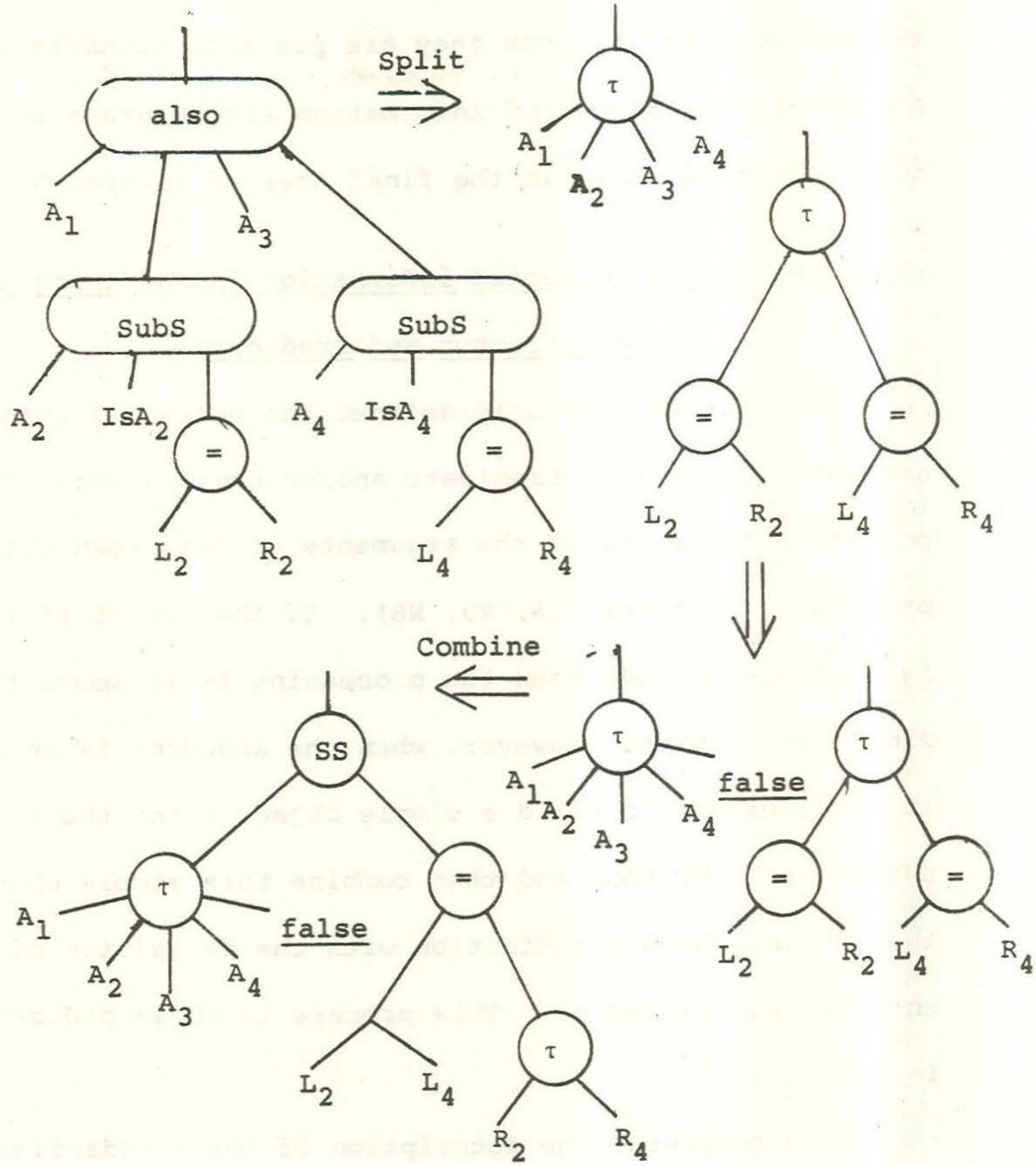

figure IV.24 Standardization of a structure with substructures. The components A_1 and A_2 of the also node should also be 3-tuples but since the other two components are nil they are depicted as single atoms for simplicity.

simply makes the selector tuple an SV node and returns it. If there are subdefinitions they are put into standard form by AD and the two pieces of information are returned as an SS node. This is shown in the final step of figure 24.

Assembling the Collected Information and Defining the Constructor and Predicate

Thus far we have only defined the packet of information necessary to build a predicate and/or constructor. This information forms one of the arguments of the named object processing routines (NS, ND, NB). If the packet of information is a simple SV node then the processing is as above for simple structures. However, when the argument is an SS node, it is necessary to build a simple object using the first tuple in the SS node and then combine this simple object into a simultaneous definition with the definition of the enclosed substructures. This process is shown pictorially in figure 25.

This completes the description of the standardizing process. While only structures were treated in detail, the processing for predicates and combined predicates and structures is similar so they will not be described further. The most important point to notice is the concurrent operation of two processes. One is collecting information for and building

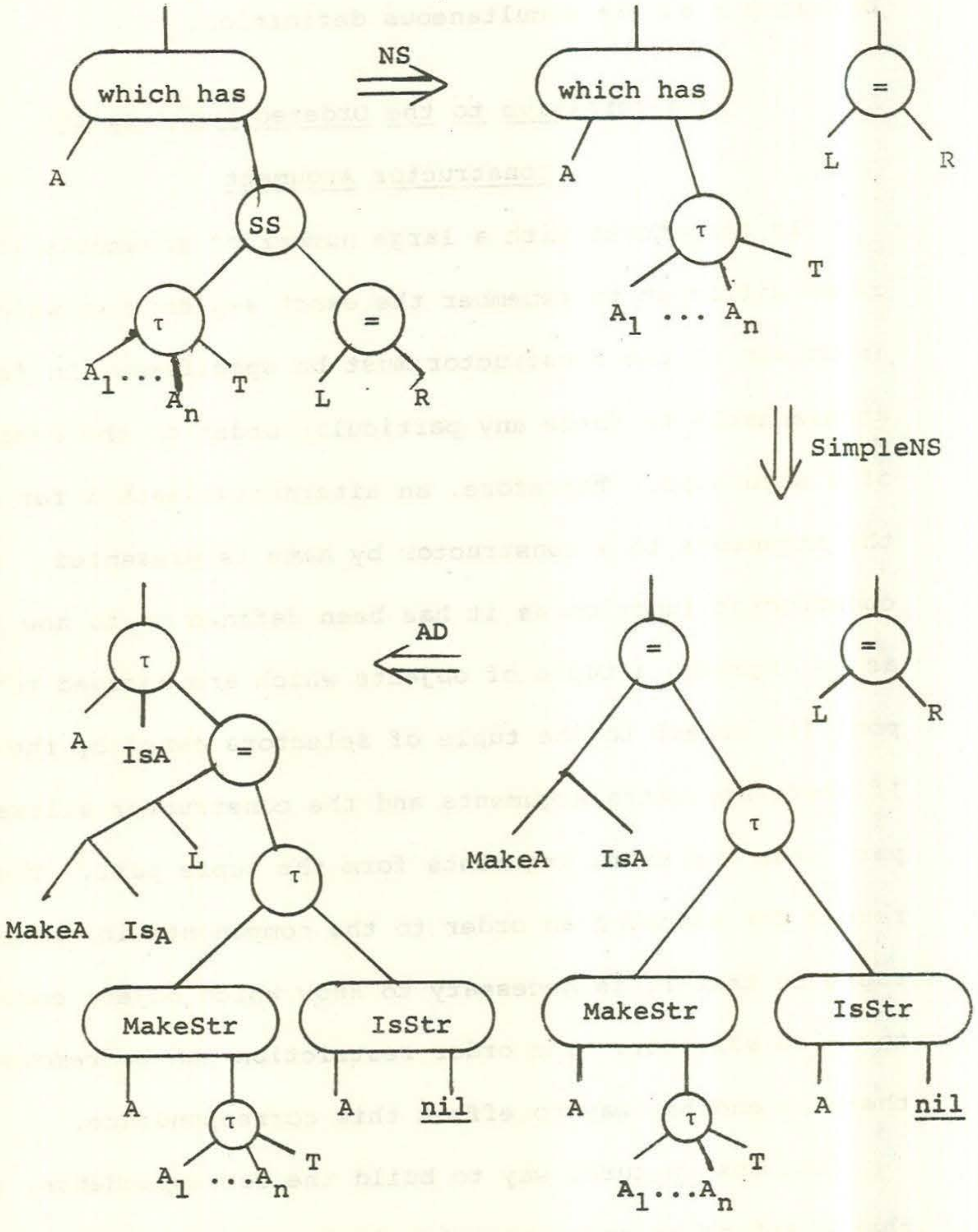

figure IV.25 Standardization of named structures with substructures.

predicates and/or structures. The other process is collecting the definitions of all enclosed structures and predicates and building a single simultaneous definition.

An Alternative to the Ordered Tuple as the Constructor Argument

In structures with a large number of arguments it is often difficult to remember the exact sequence in which the arguments to the constructor must be specified. In fact, it is unreasonable to force any particular order on the components of a structure. Therefore, an alternative method for specifying the arguments to a constructor by name is presented. The constructor function as it has been defined up to now takes as an argument a tuple of objects which are assumed to correspond (in order) to the tuple of selectors owned by the constructor. If there are extra arguments and the constructor allows a tuple part then the extra arguments form the tuple part. The only reason for assuming an order to the components in the argument tuple is that it is necessary to know which object corresponds to which selector. The order restriction can be removed if there is another way to effect this correspondence.

The most natural way to build the correspondence, given that a set of selectors exists, is to match the objects to the selectors by name. This means that it must be possible to attach a name to each object being sent to the constructor. This is done with a new object called a "name qualified value" (nqv). This object has two parts. It has a name which in the case of data structures will be an atom and it has a value which could be any object. This new object can be represented by a 2-tuple with a special tag, say nqv. The first component is an object and the second component is the name which qualifies the object.

A New Class of Objects

These new objects can now be used to build the correspondence between components and arguments. Obviously if the name of an nqv is a selector then the associated object is to be the component corresponding to that selector. An error occurs when the same name is used as a qualifier more than once in the same argument tuple. It is also an error if the name of the nqv is not in the selector set of the constructor.

These name qualified values will most often occur as components of tuples, so they should occupy approximately the same position in the syntax heirarchy as a tuple component. This suggests the following syntax:

T2::= T3 at T3 | T3

T3::=

where the first T3 is an expression which produces a value and the second is an expression which produces a name. Notice there is no need to restrict names to being atoms.

A Formalization of the Matching Procedure for Normal Values

In the syntax we have just defined there is nothing which prohibits named and unnamed values within the same tuple. Therefore, it is necessary to extend the matching procedure given above to handle mixed argument tuples. There are several possible extensions. The function Canonical in figure 26 was chosen because it seems to be one of the most flexible ones. It has two arguments, the set of selectors and the argument tuple and it produces a tuple in canonical order for that selector set. That is, it produces the tuple the user would have had to write if he hadn't used named values.

Basically it performs a two step process. The first step is to find the indices of the named components. Each name is checked against the selector set and if the name is found the index of the named component is put in a tuple at the position of that name in the selector set. That is, this tuple of name indices is sorted into the order of the selectors. Since this is basically a sorting process it is easier to describe in L-PAL although it could be written in R-PAL. The indices of the unnamed components are collected in a second tuple in

```
def  Buildvec (n,v) = S(1,nil)
          where rec S (m,t) = m eq n -> t | S[m+1,Aug t (loc v)]

def  Cstep1 (u,Sel) =
          let Chk = Buildvec(Order u,0)
             and Nam = Buildvec(Order Sel-1,nil)
             in  [Chk,Nam,Q(1,nil),u]
                where rec Q(k,Un) =
                      k > Order u -> Un
                        | Istag (u k) 'nqv' -> Q[k+1,Sort(Un,k)]
                                  | Q[k+1,Aug Un k]
                        where Sort(Unn,m) =
                           [let n = Decode(u m 2,Sel) in
                             n eq 0 or Chk n eq 1 -> undef
                                 |(Chk n := 1; Nam n := k; Unn)]

def  Cstep2 (Chk,Nam,Un,u) = R (1,1,nil)
          where rec R(i,j,t) =
             i eq Order Chk -> t
                  | Chk i eq 0 -> R[i+1,j+1,Aug t (u (Un j))]
                        | R[i+1,j,Aug t (u (Nam i) 1)]

def  Canonical (u,Sel) = Cstep2(Cstep1(u,Sel))
```

figure IV.26 The function which builds a canoical tuple. Since the named values may occur in any order, L-PAL is used to sort the indices of the named components in Cstep1. The sorted indices in Nam are used in Cstep2 to select the appropriate name qualified value for the named components indicated by 1's in the Chk vector.

the order in which they occur. The tuple Chk is used to remember which components of the canonical tuple were given by name. There is a 1 in positions corresponding to the named components.

The second step uses the two index tuples and the Chk tuple to assemble the canonical tuple. By using Chk it can tell which index tuple to use in selecting the next component of the canonical tuple. Since the named component indices are sorted and the unnamed indices are in their original order, this procedure has the affect of distributing the unnamed components into the spaces between the named components. Thus, a user need only name those components whose relative position he does not recall.

Only the indices of the components are manipulated in L-PAL to avoid losing any <u>loc</u>'s which may be in the data tuple. This approach preserves the components as they were written since Aug in S-PAL does not force any mode changes. Therefore, all <u>loc</u>'s will remain <u>locs</u> and all Rvalues will also be unchanged in the canonical tuple.

Additional Uses for Named Values

Obviously this scheme could also be used in normal function invocations. The names would then correspond to the formal parameters of the function. This would be very convenient for functions with a large number of arguments.

This name qualified value has a strong resemblance to the keyword parameters which are used in some macro systems and in various command languages. This leads to the idea of default values associated with the parameters or selectors. In the case of data functions, default values could be specified in a second tuple which was in 1-1 correspondance with the selector set. The default value would only be used when the canonical argument tuple was too short to match all the selectors. However, further consideration of this proposal is beyond the scope of this thesis.

Another use for named values that we can see is to rescind the rule which prohibits unused names in the argument list. If instead these values are just ignored, it is possible to implement the concept of "by name" assignment found in PL/I and COBOL. A "by name" assignment is a component by component assignment between two structures whose formats differ. Only those components whose qualified name is the same in both structures are changed. This could be mimicked with a by name construction which first destroys the righthand structure and attaches the appropriate selector name to each component. Then the left hand structure could be constructed from these named components.

This short section has only explored some of the possibilities for named values. Unfortunately time prevents a more thorough study.

Modifying MakeStr to Allow Named Values

It is very simple to modify MakeStr to allow named values in the argument list. The current form of the constructor produced by MakeStr expects the argument tuple to be in canonical order. Therefore, it suffices to invoke Canonical in the argument to the existing constructor. The modified form of MakeStr is given in figure 27.

```
def  MakeStr (Tag,Sel) =
        let n = Order Sel - 2
           in  Constructor
           where rec Constructor (u) =
              fn y. IsATOM y ->
                          y eq tag -> Tag
                        | y eq domain -> Buildset(Sel,t)
                        | y eq constructor -> Constructor
                        | [let k = Decode(y,Sel) in
                               k eq 0 ->undef | t k ]
                   | Sel(Order Sel) -> t(n+y) | undef
                 where t = Canonical(u,Sel)
```

figure IV.27 The MakeStr function has two embedded functions which produce functions. The first is the Constructor which is produced on applying MakeStr. The second is an anonymous function in a single variable y which is produced when the Constructor is applied. It has as "own" values the canonical form of the constructors argument tuple. Only the last line of the MakeStr definition is new in this figure.

Chapter V

A Type System for Structures

One of the major goals of this thesis is to define a system in which it is possible to build strong representations of data structures. This means that it must be possible to restrict the range of values which may be assumed by the components of a data structure. If any object may be substituted for a componant, extra or irrelevant properties could creep into the representation.

For example, consider the structure definition for an algebraic term given in Chapter IV (IV(4)). In this case a term is either a constant or a variable, or it is one of two constructed objects, a sum or a product. If the components of the sum or product could be any two objects in the universe of discourse, then it would be impossible to say much about the structure of terms beyond the fact that they have two components. In fact, the components of a sum or product are not free to be any object, but must be other instances of terms. This makes it possible to attach a very definite structure to a term. It represents the top node of a binary tree whose leaves are variables or constants and whose other nonterminal nodes are binary sums or products.

From this example it is easy to see that it is necessary to verify or validate the components of a constructed object before constructing it. Since a data structure will in general be a collection of constructed objects which are linked together in a specific way, a strong representation is possible only if the structure is validated as it is being built. The extent to which validation is performed determines the strength of the representation.

Other Reasons for Type Verification

There are two other reasons for verifying the type of components. Both of these reasons are related to optimization. If the type of an object is known or is at least restricted to some range of types, then the fact that the excluded cases will not occur can be used to improve the efficiency of a program using that object.

This type of optimization comes in two forms. The first form might be called applicative optimization because it deals with function application. Most functions have limited domains of applicability. For example, in PAL the operator "+" is not defined on tuples or functions. Therefore, it is necessary to test the operands of a function before applying it to determine if it is a legal application. If it is known that the operands are already restricted to a range within the legal domain of the function this validity test can be omitted. Hence a single test at construction time can replace many tests which would have occured when the component was used.

The second form of optimization is storage optimization. In most computers it is necessary to allocate storage space to hold values. In S-PAL the Rvalues are held in locs. If nothing is known about the range of values which might be stored in a loc, then it is impossible to pre-allocate storage. However, if the range of values is limited to a set of types with similar storage requirements, it is possible to pre-allocate storage for the loc and merely to assign the value to the existing storage. In this way storage is allocated only once instead of on every use. Both forms of optimization use the type information to compute something only once instead of many times because its value is known not to change.

Dynamic Versus Static Type Systems

There are two extremes in type checking systems. Some languages require that all type information be available at compile time and all type checking is done at that time. Examples of this type of language are PL/I, COBOL and FORTRAN. More recently languages which have no compile time type information have been developed. These languages rely on run time type checking to validate operations. PAL and APL are examples of this approach. The former kind of type checking is called static type checking, while the latter is called dynamic type checking.

As with many absolute distinctions, there are languages which are neither totally static nor totally dynamic. Typical of this class are ALGOL68 and BASEL. These languages have extensive facilities for static type checking, but allow the user to have dynamic types if he chooses. In these languages the range over which a dynamic type can vary is normally limited in any particular use. However, this is no restriction on what types may be in the range. There are language facilities for testing which of the possible types actually occur.

In the case of BASEL and ALGOL68 the type testing facility makes it possible to generate type test free compiled code even for the identifiers with dynamic types. The basic idea is to define a conditional statement which is conditional on a type test rather than on logical or arithmetic test. For example, in BASEL there is a statement of the form

when identifer is type then statement1 else statement2 end

This is interpreted as follows. First the value of the "identifier" is tested against the "type". If the type matches, then "statement1" is executed; otherwise, "statement2" is executed. However, the difference here is that for the duration of "statement1" (which could be a group of statements) the type of "identifier" is known to be "type". Hence, the generated code for "statement" can be free of type tests on "identifier". If "statement2" is executed then nothing is known about the type except that it is not "type". It may be the case that "statement2" is another when statement.

Why are both dynamic and static type systems necessary? Even though static type systems allow greater optimization of the generated code they do so at the cost of flexibility. The static type systems perform early binding on the range of values on identifier may denote. There are cases, such as data structures like TERM (IV(4)), where an object may have one of several alternative forms. It is therefore necessary to be able to determine for each instance which form actually occurs. That is, the binding of the type must be delayed until run time. This implies that some form of dynamic type checking is necessary.

The Simplicity of Dynamic Type Checking

There is a second reason for the popularity of dynamic type checking. It is in general a much simpler task than static type checking. With dynamic type checking the value to be validated is known. Therefore, type checking is just a question of set membership. With static type checking the particular value is unknown and only the range of attributes the value may have are known. Therefore, it is necessary to test if the set of objects whose attributes are known is contained in the set of objects that are valid. This changes a question of set membership into a question of set containment.

The set containment question is in general much more difficult to answer. For example, consider the context free languages. It is possible to decide if a word ω is in an arbitrary context free language, but it is undecidable whether an arbitrary regular set is contained in a context free language. Thus, the set containment problem is seen to be more difficult then the set membership problem, and a static type system will need careful specification of the range of values in a type class.

A Type System Based on Predicates

The type systems presented in this chapter is suitable only for dynamic type checking. The primary reason for this is that static type systems are much more difficult to construct. In fact, even the limited goal of a dynamic type system is not particularly simple to achieve as we will see below.

Verifiers in Structure Definition

As we noted in Chapter IV, Landin included more than the selector set in his structure definitions. He also included type information with every component selector. That information was primarily descriptive. It tells the reader what to expect as a value for that component. For example, the declaration of "λ-closure" has three components.

A λ-closure has

a bound variable part which is a variable

(1) and a λ-body which is an λ-expression

and an environment which is an environment

The phrases beginning "which is" describe the type of the component. Notice the similarity with the predicates of S-PAL.

When we consider constructing such objects it is easy to see that the type information can be used to verify that the intended components are indeed of the correct form. The type checking can be done dynamically when the components are presented to the constructor. In this case, type checking consists of testing the objects in the constructor argument for the properties required of the corresponding component. This can be done in general by a predicate function which tests the required properties and returns <u>true</u> or <u>false</u>. If the results of all the component tests are <u>true</u>, then the argument is suitable and the construction is done. If any component test fails, then the construction is aborted.

The Concept of Type in S-PAL

The above discussion leads to a natural definition of type in S-PAL. A type is a predicate, usually called a verifier. As used here a predicate means a function of one argument which is defined over the universe of discourse and which for every object yields a value true or false. Those values for which it yields true are said to have the type it defines.

This is a very general concept of type. It includes tests for the simple built in types such as integer, real and character using the built in predicates IsINTEGER, IsREAL and IsCHAR. Hence, it includes the normal concept of primitive type. It is also possible to perform complex tests which define such types as "binary trees of depth less than or equal to n". Such a predicate would have to know the representation of the tree and could scan the tree to check the depth condition. The major problem with defining types this way is that it is too general. This will be discussed in greater detail in the sequel.

The Syntax for Verifiers

While it might be possible to restrict the verifiers to previously defined S-PAL predicates, there are times when this is inconvenient. In fact, there are times when the type must be defined simultaneously with its use. For example, consider the definition of LIST (IV(6)).

def rec LIST which

(2) is(HEAD which IsLIST else IsATOM

also TAIL which IsLIST else IsATOM)

else IsNIL

In this definition both the HEAD and the TAIL component have the same verifier. It is an unnamed predicate which yields true for any ATOM or alternatively for another instance of LIST. Because rec was used, the instance of IsLIST in the verifier definition refers to the predicate IsLIST defined by the standardization of the LIST definition. However, the predicate IsLIST only checks the tag of a structure for equality with "LIST" (See Chapter III). It does not make tests on the components which would cause itself to be invoked again. Hence, the recursion always terminates after one step. IsLIST will also yield true if the argument is nil since IsNIL is given as an alternative type for a LIST.

This definition of LIST was written with an explicit rec. This is consistent with the general PAL philosophy which requires rec to be written for all recursive functions. However, in S-PAL it was decided not to use rec in structure definitions, but rather to assume an implicit use of rec in all structure definitions. It should be emphasized that the implicit use of rec was not done to make it more convenient to define self referential structures.

The reason rec is implicit in a structure definition is that it is a simple solution to the problem of needing to use a single predicate definition in two different places. When a structure contains a substructure (or subpredicate), the predicate associated with that substructure becomes the verifier for the component represented by the substructure. However, that predicate must also be given an external name so that it is accessable to the user. Since these two uses of the predicate definition occur at different places in the standardized tree, the same copy of the predicate construction cannot be used in both places.

This problem admits to two solutions. First, two copies of the predicate definition could be made. Then one copy would become an argument to MakeStr for the definition of the enclosing structure and the second copy would be bound to the subpredicate's external name. However this solution has two disadvantages. The process of copying the definition is messy to specify formally and it involves unnecessary replication of information.

A much better solution is to give a name to the predicate and to use that name to refer to the predicate from both places in the standardized tree. This name can be an arbitrary local name for the predicate which is only defined on the righthand side of the simultaneous definition for the whole structure. This name would never be accessable to the user, but would be used in place of the predicate construction in the argument to MakeStr and in the definition of the external name.

Unfortunately the process of defining such local names greatly complicates the already complex standardization process. Furthermore, at the cost of making every structure definition implicitly recursive it is possible to use the external name of the predicate instead. This name must be defined anyway and with the implicit rec it can be used in the MakeStr argument to identify the verifier. Therefore, the simpler solution to the problem was chosen. This solution does not require any changes in the standardization process except those required to build the tuple of verifiers for the constructor to use. Without the use of rec the external name used in the verifier tuple would be undefined or would refer to some previously defined name.

This problem is not just restricted to substructures. It also occurs when subpredicates are defined in either structures or as alternatives in predicate definitions. Therefore, this solution is also needed with the typeless structures defined in Chapter IV.

The Syntax for Verifiers

Because types are restricted to structure definitions, the syntactic additions are very simple. The definition of <selector> is extended to include a verifier, as is the tuple option on the <annonymous structure>.

<anonymous structure>::={<selector>also$\}_1^\infty$ <selector>

| {<selector> also$\}_0^\infty$ tuple <anonymous predicate>

| only <selector>

<selector>::= <atom>

| (<named structure>)

| <named predicate>

or in the abbreviated form

S1::= {S2 also$\}_1^\infty$ S2 |{S2 also$\}_0^\infty$ tuple P1 | only S2

S2::= ATOM | (S) | P

The <anonymous predicate> in the tuple option is the verifier for every component of the tuple part. The <named structure> is still interpreted as a substructure definition but in addition, the predicate it defines becomes the verifier. The <named predicate> form will be more common. It defines both the selector and the verifier. The unqualified name of the <named predicate> is the selector and the predicate it defines is the verifier.

A new interpretation is given to the ATOM occuring alone. This defines the selector name as before. However, it is also used as the base on which the name of a predicate is constructed by prefixing "Is". It is assumed that a predicate of that name has been previously defined. For example, if the user wished to build the TIME structure (IV(10)) without qualifying the substructures he might use

```
              def ID which has
                     ID_NO which IsINTEGER
                     also NAME which IsCHAR
    (4)
              def HOURS which has
                     WORKED which IsREAL
                     also SICK which IsREAL
                     also VAC which IsREAL
```

to define the substructures and then define TIME by

```
    (5)       def TIME which has ID also HOURS
```

In this case the verifiers for ID and HOURS are the predicates IsID and IsHOURS defined in (4).

This new syntactic extension requires only a small change to the abstract syntax. It allows S2 to be rewritten as P as well as ATOM and S. The abstract syntax tree for the definition of ID given in (4) is shown in figure 1. The changes required in the standardizing process are a little more complex as can be seen from the standardized tree for ID which is given in figure 2.

There are two things which increase the work of the standardizing routines. The primary addition is to use the tuple of predicates returned by Split as the verifiers for the corresponding selectors also returned by Split. These two tuples are combined with the tag name to form a 3-tuple which is the argument to the extended version of MakeStr described below.

The other addition is slightly more complex. The problem which it solves arises because not all predicates are given names. In particular, predicates defined solely as verifiers remain anonymous. This is a result of a design decision to avoid proliferating names when they had no apparent use. This means that the MakeStr function cannot reference the verifier by name as described above for substructures, but instead must use the predicate construction directly. It also means that the generation of names in such standardizing functions as NS, NP and NB must be controlled.

The solution to this problem is to identify the contexts in which names are and are not generated. Then an additional

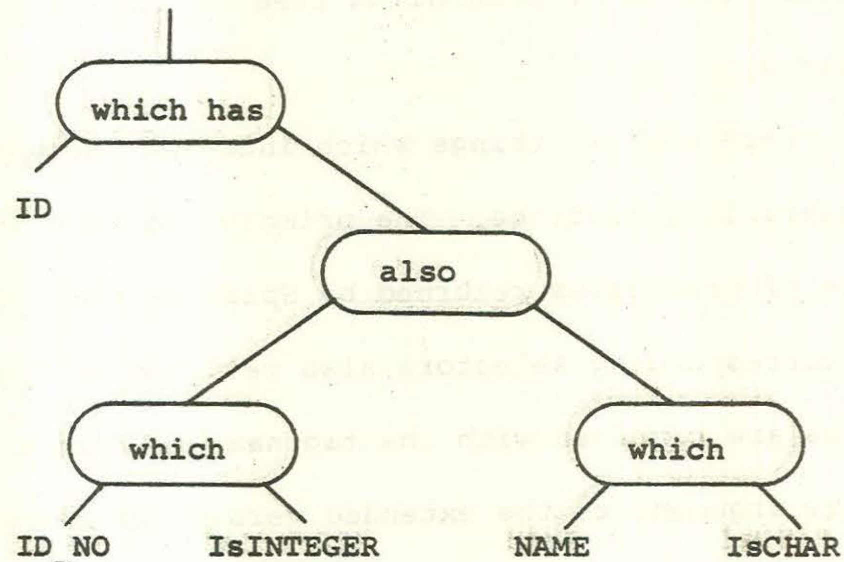

figure V.1 Abstract syntax tree for the structure ID in definition (4)

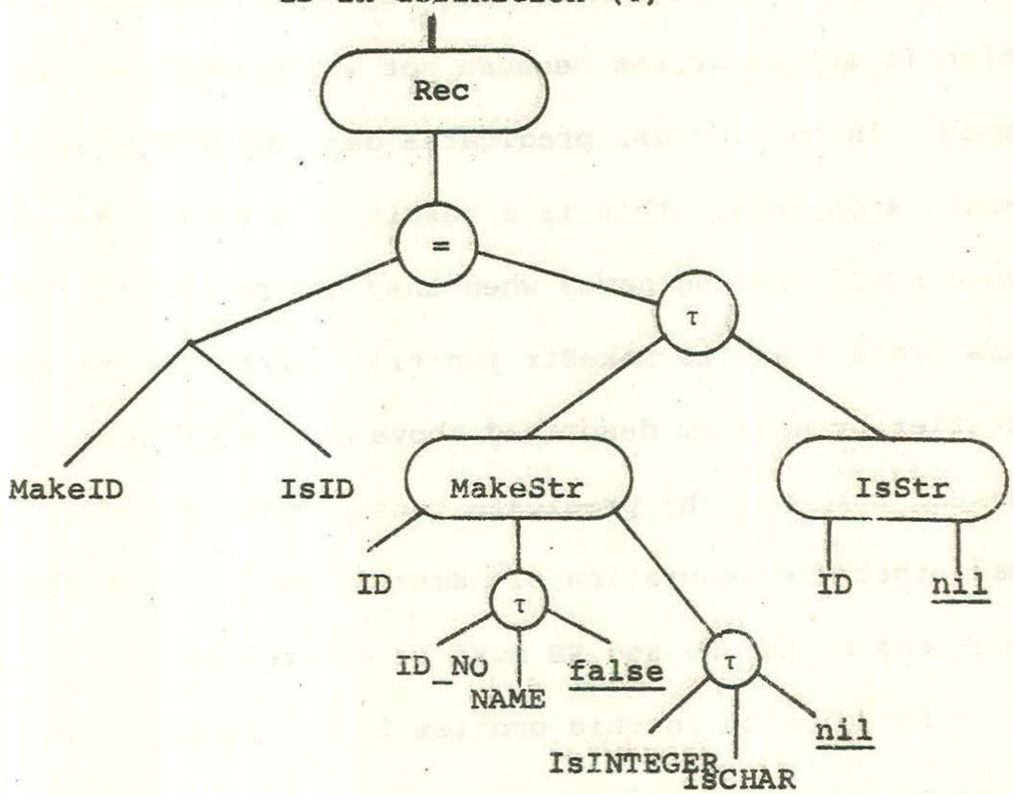

figure V.2 The standardized tree for the structure ID in definition (4) argument can be added to the standardizing routines to carry the context information. Before specifying the contexts it is necessary to define several terms carefully. We will use the term abbreviated definition for the definition which defines both a constructor and a complex predicate. An example of an abbreviated definition is the definition of LIST given above (2).

We will say that a definition is immediately contained in another definition if there is a path in the abstract syntax tree connecting the two structure, predicate or abbreviated definitions and if there is no other structure, predicate or abbreviated definition on that path. For example, the definition of ID in Chapter IV.(10) is immediately contained in the definition of TIME. We will use the term contained when we only require that there is a path between the two definitions in the abstract syntax tree.

The contexts for name generation can be described as follows.

1) A structure, predicate or abbreviated definition which is not contained in any other structure, predicate or abbreviated definition is said to be in a type 1 context. In this case the unqualified tag name is used as the base name for generating the constructor and predicate names. In the above examples (4)

and (5) the structure definitions for ID, HOURS and TIME are all in a type 1 context.

2) A structure, predicate or abbreviated definition which is immediately contained in a type 1 predicate definition or another type 2 predicate definition, is said to be in a type 2 context. In this case the unqualified tag name is also used as the base for the function names. However, the name of the predicate is also used as an argument to IsStr in defining the predicate for the immediately containing predicate definition. The definitions of SUM and PROD in example (4) of Chapter IV are structure definitions in a type 2 context.

3) A structure or abbreviated definition which is immediately contained in a type 1, 2 or 3 structure or abbreviated definition is said to be in a type 3 context. As we noted in Chapter IV, the name base of such a definition is made by qualifying the tag name with the tag names of all the structures in which it is contained. This qualified name is then used to define the external names of the predicate and constructor. The name of the predicate is also

used to represent the verifier for the corresponding component of the immediately containing structure. In the definition for TIME in Chapter IV, example (10), the structure definitions for ID and HOURS are in a type 3 context.

4) A definition is said to have a type 4 context if it is either

a) a predicate definition which is immediately contained in a type 1, 2 or 3 structure definition, (i.e., it is a verifier definition)

or

b) a predicate, structure or abbreviated definition contained in a type 4 definition.

In either case, no name is created for the object being defined. Instead the constructed predicate is used directly as the verifier for the corresponding component of the immediately containing structure. The predicate "HEAD _which_ IsLIST _else_ IsATOM" in the definition of LIST in example (2) is in a type 4 context.

The context information is passed from level to level as the abstract syntax tree is standardized. It begins with a type 1 context and the argument is modified in US, UP and AP to establish the correct context for the components of these anonymous objects. The information packet (selector, predicate and lower level definitions) is built in NP, NS and NB which use the context information to decide whether an external name is defined. These routines also decide whether the predicate component of the information packet is the name of the predicate or the result of applying IsStr. Other than this the processing is basically the same as that described in Chapter IV. The complete gedanken interpreter for S-PAL with typed components is given in Appendix B.

Using Verifiers in the Constructor

One of the main advantages to defining types by predicates is the simplicity of the validation process in the constructor. It is performed by applying each component predicate in the verifier tuple to the corresponding component of the data tuple. The results of the individual verifications are anded together to produce the combined result. If the result is <u>false</u>, the constructor returns the special value <u>undef</u>. Otherwise, the constructor produces a data function defined on the components of the argument tuple.

The version of MakeStr with verification is given in figure 3. It uses an auxilliary function Verify to validate the components

```
def  Verify (V,t) = Q(1,true)
          where rec [ Q(k,Tv) =
                          k ge Order V -> Isnil (V k) ->Tv
                                                       | R(k,Tv,V k)
                                         | Q(k+1,Tv & V k (t k))
                 and R(m,Tv,Vr) = m > Order t -> Tv
                                         | R(m+1,Tv & Vr (t m),Vr) ]

def  MakeStr (Tag,Sel,Ver) =
        let n = Order Sel -2
           in  Constructor
           where rec Constructor (u) =
              not Verify(Ver,t) -> undef
                |{fn y. IsATOM y ->
                                    y eq tag -> Tag
                                  | y eq domain -> Buildset(Sel,t)
                                  | y eq constructor -> Constructor
                                  | [ let k = Decode(y,Sel) in
                                           k eq 0 -> undef | t k]
                             | Sel(Order Sel) -> t(n+y) | undef }
                      where t = Canonical(u,Sel)
```

figure V.3 The Makestr function which verifies the component values. The only change is to make the result of the constructor conditional on the verification of its argument. If the argument is not verified then the result is undef, otherwise is is a data function as before. The argument is put in canonical form before the verification.

of the canonical form of the data tuple. The only complication in **verify** is the processing of the tuple part of a mixed domain structure when it is present. If the final component of the verifier tuple is nil then no tuple part exists so the truth value is returned. If, however, the final component of the varifier tuple is not nil then it is the verifier for all the components of the tuple part. In this case the tuple verifier is applied to every component of the tuple part and the results of these applications are combined with the results of the symbolic part to give the function result.

Because a tuple has a variable number of components it is not possible to specify individual types for more than a fixed initial segment of the tuple. Therefore, it is necessary to define the types of the components in a manner which will allow arbitrary extensions of the tuple. One way to do this is to provide a function which given the index of a component, would produce the verifier for that component. This would allow a wide variety of mixed types in a tuple. For example, it would be possible to describe a tuple in which the even components were real numbers and the odd components were their character representations.

However, this approach to types is probably more powerful that is really needed. In general, fancy combinations of types do not occur in tuples. This is particularly true when the data functions are included since most mixed type structures are easier to define in terms of symbolic selectors. Therefore, we chose to limit the types of tuple components to a single verifier which validates every component. This is consistent with most other programming languages. If the user wants to mix types, he can use a verifier which will accept several alternatives or he can use the verifier IsANY which always returns true. This latter verifier allows him to construct tuple parts which are like the unverified tuples of the current PAL system.

This completes the description of the representation for data functions. The complete set of programs is collected together in Appendix D. While there are a large number of auxilliary functions used in defining MakeStr most of them are used only during the construction of data functions or for the special selectors. Therefore, a simple data reference is reasonably efficient.

The Problems Associated with Unrestricted Verifiers

Even though we have restricted S-PAL to dynamic types, there are a number of problems which arise in checking types. The most obvious of these is the handling of locs. All other objects have a fixed Rvalue. Thus, it is sufficient to test that Rvalue at construction time to verify that the component it occurs in is correct. However, the Rvalue associated with a loc can be changed by an assignment statement. This means that verifying the appropriateness of the Rvalue contained in the loc at construction time is insufficient to insure the validity of that component at later times.

There are two solutions to this problem. One solution is to make the problem disappear by treating all locs as a single indistinguishable type. In this case the verifier would only check whether or not the component was a loc. Because the constructor binds the data function to its components, a component which is a loc will remain a loc forever. Hence, the verification is valid at all times after the construction.

The other solution is to attach a type predicate to the location. This predicate would be used to verify the validity of any assignment. Then as long as this predicate is at least as restrictive as the verifier for the component whose value is the loc, all valid assignments will also satisfy the verifier. Hence, this construction is also valid at all later times. The properties of these solutions are developed in detail below.

Treating all locs as a Single Type Class

Certainly the locs form a type class because they are obs and there is a predicate IsLOC which distinguishes them.

However, the idea of this solution is to prevent locs from occuring except where a loc was explicitly indicated in the structure definition. That is, a loc containing a real number would not be a valid component for a verifier which requires a real number. This solves the validation problem by preventing all updates when an Rvalue typed object is required by the verifier. Conversely if a loc is allowed as a possible value of a component then no other type checking is performed on that component. Therefore, the value of the loc may have any type except loc.

This means that the only way to build a structure with a strong representation is to build it solely from Rvalues. This would appear to prevent updates to structures with a strong representation. Actually, it is possible to perform a limited form of updates and still have proper validity checking. It is possible to decompose the structure into a tuple, update a component, and rebuild the structure from the updated tuple using the constructor obtained from the original structure. If the structure was the value of a loc then the updated copy can be made accessable by assigning it to that loc. Because the same constructor was used to build the new structure the updated component must satisfy the same verifier as the original component. Hence, the strength of the representation is unchanged.

The generalized update operator Update is given in figure 4. The auxilliary function Index is used much like Decode (See Chapter IV) to get the index of the component of the data tuple to replace. If the value is zero then no such component exists and no update is done. Otherwise, the function Insert is used to decompose the data function and replace the component to be updated. The constructor obtained from the original structure is used to construct a new data function on the updated tuple. Note that all the components of the new data function, except the updated component, share with the components of the old data function. Hence, this function acts much like the function AuG. (Chapter IV, figure 17)

Thus, we see that this solution is practical and even allows most of the operations that one would want to perform on a data structure. The only real problem occurs when the structure to be updated is referenced as an Rvalue in some other structure. In this case there is no way to update the structure and preserve the sharing.

Shaped Locations

The alternative to limiting type checking on locs is to make the locs check the values being assigned to them. This can be done by attaching to each loc a predicate which is used to test whether or not an assignment is valid. If the value being assigned satisfies the predicate, the assignment

```
def  Update (D,s,v) =
          let C = D constructor
            and i = Index(s,D domain)
              in i eq 0 -> D | C(Insert(D,i,v))

def  Index (s,t) = R(Order t)
         where rec R k =
                          k eq 0 -> 0
                          | s eq t k -> k
                          | R(k-1)

def  Insert (D,i,v) =
         let A = D domain in Q(1,nil)
            where rec Q(k,t) =
                          k gt Order A -> t
                          | k eq i -> Q(k+1,Aug t v)
                          | Q[k+1,Aug t (D (A k))]
```

figure V.4 The Rvalue update function. The Index function yields the index of the data tuple component to replace. The Insert function decomposes the data structure into its components replacing the component to be updated. This new data tuple is then used to construct the new data function returned as the result of Update.

is valid. Otherwise, the value is rejected and the assignment is aborted and an error message is given. This action is similar to what happens when an operator such as "+" is applied to a data object, such as a character string, for which no result is defined. This also produces a run time type error. The locs with attached predicates will be called shaped locs because only values of the correct type (shape) can be assigned to them.

The only problem with this solution to the validation problem is that it is necessary to insure that the predicate attached to the shaped loc defines a type class contained within the type class defined by the verifier the loc must satisfy. There are two solutions to this problem; each has a different disadvantage.

It is possible to ensure that the predicates of component locs are consistent with the verifiers for these components by creating the locs as the structure is built. These created locs would receive the verifier as their attached predicate. Therefore, only legal assignments would be allowed. In general, a new loc would be created for every component of the data tuple which is a loc. Then the Rvalue of each original loc would be assigned to the corresponding new loc. The assignment and hence the construction would only be done if and only if the Rvalue was of the correct type.

This has the advantage that it is not necessary to insure that the domain of the original shaped loc is contained within the domain of the verifier. The only requirement is that the current Rvalue be within the domain of the verifier. However, it has the disadvantage that it is impossible to create data structures which share locs.

If the sharing of locs is to be allowed a different solution is needed. In this case it becomes necessary to be able to decide when the predicate on an existing location defines a type class that is contained in the type class of a verifier. As we have already remarked, this problem is in general undecidable. Thus, loc brings us back to the set containment problem we sought to avoid with a dynamic type checking system. However, this seems to be the only reasonable solution to the problem of shaped locs in structures.

Why Restrict Shaped locs to Structures?

If we allow shaped locations in structures then why not allow them anywhere in S-PAL. There is certainly no reason to restrict them solely to structure definitions. In most places in the language the problem of checking set containment doesn't even arise. It also has the advantage that it makes assignment more like the other operations in the language.

Since "+" will raise an error when its arguments are mismatched, it is reasonable to expect the assignment operation to fail when its type constraints are not met.

There is some question as to what objects should be given types. Should they be restricted to locs and the components of structures or should they also be definable for other linguistic features such as names. In most languages it is possible to give type restrictions to formal parameters which are really only dummy names. They are bound to values only when the procedure in which they occur is called. At that time the type conditions could be verified and the calling argument rejected if the type test failed. In BASEL, which allows variable bindings, all names can be given types which will be verified when the name is bound.

The main problem with typed names is that the set containment problem occurs again. Since a name may be bound to a location, it is necessary to ensure that the type of the location is consistent with the type of the name. This is, in particular, a problem with parameter names in procedure calls. In any case it is not too difficutt to visualize syntax for typed names which is similar to the S-PAL predicate syntax. In addition the loc operator would have to be extended to make it possible to create shaped locs. Thus, we see that very little extra work is required to extend the type facility to the whole language once it is defined for locs in structures.

Function Types

The locs are not the only obs which cause problems in the type system. Functions are also difficult to handle. When a function is a component of a data structure, it is necessary to verify that the domain and range of the function are valid. This is, of course, undecidable in general.

One way to solve this problem is to embed the component function within a checking function. This checking function first tests its arguments to see if they conform to the types allowed by the verifier. If they do, they are passed on to the component function. When it returns, the checking function makes sure the result is in the correct range and if so returns it. The operation of embedding the component function in a checking function is called projection by Reynolds [31]. The problem with this approach is that while it guarantees that nothing outside the domain and range will work, it does not ensure that the component within the projected domain and range.

An alternative to the projection function is to require every function to have a description of its range and domain in terms of a very simple language. For example, the language of regular expressions might be appropriate. Then the domain and range condition could be evaluated by checking these descriptions. The language must be simple for otherwise it is impossible to test for the equality of two different descriptions.

The S-PAL Solution

The problems discussed above are just some of the more obvious complications that result when types are defined by unrestricted predicates. For example, it is undecidable when two alternatives in a predicate definition define intersecting type classes. Therefore, it would appear that the appropriate solution to the problems defined above would be to define restrictions on the predicates which would make questions such as set containment answerable. This would make it possible to solve the problem of strong representations which included *locs* by using shaped *locs*.

Unfortunately the design of such a type system is beyond the scope of this thesis. Some steps in this direction can be found in the work of Morris [25], Reynolds [31], Jorrand [12] and van Wijngaarden [37]. But designing a type system which provides for static type checking, but is not too restrictive, is still an open problem. Therefore, we choose to allow the user the ability to use any predicate as his verifier.

This makes it possible for him to solve the above problems. He can ensure strong representations by putting a test for *loc* in the predicate for every component. If the component should be a loc, this predicate would check to make sure the component is a loc. If the component should be an Rvalue the predicate would check to make sure that a loc did not occur. The problem of checking functions is more difficult.

One solution is to make every function which could be assigned to a component provide descriptive information when a special argument is given. This is analogous to the information provided by the data functions when they are applied to a special selector. This information could be used in the predicate to accept or reject the function.

These solutions are not as pleasing as a suitably restricted type system and shaped locs. In particular, they put most of the work in doing type checking on the user. However, allowing the unrestricted predicates provides the generality needed to define different type constraints. This seems to be the best solution when no particular type system is accepted by everyone.

Chapter VI

Conclusions and Analysis

The preceding chapters have presented a data structuring facility for PAL. This facility makes it possible to describe the nodes of a data structure in a natural way. It provides a wide range of possibilities for connecting and referencing these nodes. In particular, it makes PAL more flexible and gives the user greater control over the form and processing of his data. In this chapter we summarize the salient and novel aspects of S-PAL, we discuss a possible implementation and we also discuss possible directions for extending this work.

Treating Locations as Values

Locations or Lvalues should be obs. It does not seem useful to isolate the loc from the other values in the system. It shares many properties with other values. For example, it can be the result of a function, used as an argument to a function, used in an expression, etc. It also has some special properties which other obs do not have. For example, the value of the left hand side of an assignment statement must be a loc. However, addition is only defined for integer or real values. Thus, other values have special properties too.

Another reason which is given for the special treatment of loc is that there are no location constants. However, there is at least one very reasonable interpretation for a location constant. In BCPL [29] and other languages there is the concept of a global variable. In BCPL this is a variable which is located in a vector which is external to every block of the program. This variable can be referenced from any block by declaring the name to be global to that block. Then any reference to that name will refer to the unique copy in the external vector no matter what names are defined in the environment of the block. This is similar to the EXTERNAL variable of PL/I. These variables are often the only way for separately compiled procedures to share values.

The natural way to implement this feature in S-PAL is to introduce location constants. A location constant is a name for a particular location which always designates the same location no matter in what environment it is used. That is, two location constants designate the same location when and only when their representations in the concrete syntax are identical. They can be viewed as locs with explicit addresses. Because they always designate a unique location, they serve exactly the same purpose as the global variable in BCPL.

Perhaps the most important argument in favor of making locs a class of obs is the flexibility it adds to the language. It gives the user control over how the names he defines will be used. He can prevent the misuse of the assignment operation by binding names to Rvalues and building structures with fixed links. He need only use a loc when he wants to be able to modify a value. We conclude that the benifits of treating locs as obs outweigh any disadvantages.

Functional Data Structures

There is a very definite need to be able to describe the structure of a data element in terms of mnemonic component names and without forcing an ordering on the components. In S-PAL this facility is provided by the introduction of data functions. This represents an extension of the ideas of the PAL tuple and the functional data structures of GEDANKEN. In particular, the domains of the data functions were extended to include symbolic selectors in the form of atoms. These atoms, like integers, are constants with a fixed value which is independent of the environment in which they are used. Therefore, the data functions can be saved on a secondary storage device and used by other programs.

We have defined a particular syntax for defining a constructor and predicate for a data structure. This makes it easy to define a set of data functions and it documents their format. However, we do not restrict the class of data functions to the results of the constructors produced from the structure definitions. We intentionally defined the class of data functions as those functions which produce the correct information when applied to the special selectors tag, domain and constructor. Therefore, if the user cannot express his data elements in terms of a structure definition he can always write his own data function.

The special selectors were chosen as a useful set of attributes that every data structure should make accessible. We have given examples which show how these attributes are used. However, we do not claim that these attributes are necessary or sufficient for characterizing data structures. Our only claim is that the attributes we chose appear to be present in every data structure and making them available makes it possible to define very general operators on the class of data functions.

A Type System Based on Predicate Functions

A type system is a necessary part of any data structuring facility which provides for strong representations of the data. This is perhaps the weakest aspect of S-PAL because the type system we chose does not allow static type checking. In fact, due to its generality, the relationship of two arbitrary type classes is undecidable. However, the novel approach of defining a type class by an unrestricted predicate function provides the user with a very flexible concept of type. He can define very restrictive type classes by writing very complex programs which test a wide variety of conditions. Alternatively, he can use the predicates created by structure definitions or the built-in primitive predicates when only the general range of values of an object is important.

The predicates defined by structure or predicate definitions are very elementary. They will accept any data function which returns the correct tag or which satisfies the alternatives of a predicate definition. This definition of type was chosen because it seems to be the simplest condition which defines a set of data functions of the same type. The user may use the other information provided by the special selectors to define more restrictive type sets.

One of the main uses for the type information is to distinguish several alternative data structures which might occur in a particular context. In most cases, each of the different data structures is processed in a different way. In the current PAL the proper processing code is selected by using a sequence of conditional expressions. This is inefficient since it requires that a sequence of tests be made to find the correct processing code.

It is more efficient to use the type value to select the correct code directly. Since the tag is a single value which represents all the type information, it is possible to use it to determine which one of a set of expressions is to be used in processing the data structure. Each possible tag value would be associated with an expression which would process the data structure with that tag. Then the multiway choice would be evaluated by executing the expression whose associated tag matched the tag of the data structure.

This is a generalization of the conditional expression which removes the need for sequentially testing the type to find the right expression to use. It, therefore, can be implemented by techniques, such as hashing, which make it possible to choose the processing expression with only one type test. This facility can be generalized to allow multiway choices on any value, not just tags. It is similar to the case expression in ALGOL 68 or in a statement form to the switchon statement in BCPL. The ability to use this feature is one of the main reasons that tags are values in S-PAL and are included in every data function.

A Possible Implementation for Data Functions

It would be unreasonable to propose an extension for data structures without giving some thought to the implementation of those structures. Since data structures in S-PAL are represented by functions, it would seem natural to implement them as functions. In fact, in the general case there is no other alternative. However, the data functions created in structure definitions, let us call these SDDF's, have many more properties than an arbitrary data function. They are all represented by variations on the same function which is produced by the constructor created by MakeStr. In fact, the only parts of the function which vary are the data tuple, the tag and selector set.

This suggests that it is only necessary to store the varying parts with each instance of the SDDF. A special type code can be stored with the varying parts to indicate that the standard SDDF accessing function is to be used to access the information. In fact, the tag and selector set only vary among SDDFs with different types. They are constant for different instances of a single type of SDDF. Hence, every instance of a particular type SDDF could refer to the same tag and selector set information.

Therefore, we propose that SDDFs be represented like tuples with an extra component. In the current implementation of PAL a tuple is represented by a type code and a list of pointers (addresses) to the component values. Hence, the SDDF would be represented by a type code identifying the value as an SDDF, a zeroth component which is a pointer to the selector set and tag, and a list of pointers to the components of the data tuple. This internal representation is just as efficient as the current PAL representation for structures which consists of a tuple of data components with an extra component to hold the tag.

This defines an internal representation which uses storage efficiently. However, MakeStr is a complex function with several auxilliary functions so it is not clear that the construction and use of data functions are also efficient. Actually, most of the complexity of MakeStr is in the construction of the data function. The data tuple must be put in canonical form and verified. While these functions are necessary they are used only once for every instance of a data structure. Note also that with this representation the canonicalization can be done by permuting the list of pointers in the data tuple. It is necessary to create a copy of the pointers to the values in the argument tuple because that tuple cannot be modified. Hence, very little extra work is required to create the copy in the canonical form.

In a reference to a created data structure only the Decode function is used. This was specifically isolated so that the lookup processes for converting symbolic names to integers could be done by hashing or if an associative memory is available by associative lookup. In the cases where the data function is applied to an atom directly, the decoding process can be performed at translate (compile) time. Then the resulting integer can be used to select the correct component of the SDDF at run time. This conversion to a relative offset in the SDDF tuple can save a lot of time if the selector is frequently used.

Possible Modifications to S-PAL and Future Directions

In the preceding chapters we have compared S-PAL with various aspects of other languages. These comparisons were directed at language features that are in both S-PAL and the other languages. In this section we wish to explore some of the language features of these other languages which are not in S-PAL. These are candidates for possible extensions or modifications to S-PAL.

Allocation and Initialization

One major deficiency in S-PAL is the lack of control over the allocation of data. Since all data is not used in the same way, it is possible to perform more efficient storage management if the data is separated into classes with similar storage utilization. For example, ALGOL 68 defines two classes of storage. There is local storage which is allocated in a stack and is released whenever the procedure in which the storage was allocated terminates. There is also global storage which is allocated from an amorphous collection of storage called the heap. As the name indicates values allocated in the heap are retained as long as there is a reference to them. Therefore, the heap must be garbage collected. It is obvious that by having the user separate out the storage which can be allocated with a stack discipline, the heap is exhausted less frequently and, therefore, fewer garbage collects are needed.

PL/I has an even larger set of storage allocation classes. It has both implicit stack storage (AUTOMATIC) and explicit stack storage (CONTROLLED). It also has a set of classes called areas. These are to the heap what named common is to blank common. These named regions are all distinct and storage can be allocated from anyone. One use for multiple areas would be to have different storage control mechanisms. Storage allocated in one area might have use counts while storage allocated in another would be garbage collected. Another use for areas is to give a name to a data base that was allocated in that area. It could then be saved with a single area I/O statement. Areas provide a great deal of flexibility in the storage allocation process.

Control over the allocation of storage could be added to S-PAL by providing a new argument to the constructor function. This argument would specify the space from which the data function should be allocated. However, there are many problems to solve. For example, is only the SDDF allocated in the specified space or is it necessary to copy in the values it points to. If so, how far does such a copy go. It also might be convenient to add an extra special selector which would produce the name of the space in which the data function resides.

Initialization is almost always linked with allocation. The reason is that it is impossible to initialize something before it is allocated and it must be done before the object is referenced. However, there are times when it becomes necessary to delay initialization. For example, when a ring structure is being created it is only possible to initialize pointers to previously created nodes. Therefore, the ring can only be closed after all the nodes are allocated.

The problem in S-PAL is that the only way to delay initialization is to use a loc. For example, the above ring could be closed by assigning a reference (1-tuple) to the last node to a loc in the first node. This type of initialization was one of the uses for Standish's constructor modifier. However, it should be possible to close the ring with a permanent, non-updatable link. Therefore, we might include a new type of loc which acts as a place holder for an unresolved value. This loc could be updated as above but the first update would replace the loc with a permanent connection to the value which was assigned. This might be called a one shot loc. This would allow delay initialization to values that were not again updatable.

Load-Update Pairs and Implicit References

There is a basic and disturbing assymetry to S-PAL. It is possible to replace every reference which loads or uses a value with a function which calculates the value. However, it is not possible to replace the lefthand side of an assignment with a function which decides how to store a value. To solve this problem it is necessary to introduce a generalization of the Lvalue. This is called a Load-Update Pair (LUP) by Strachey[35] and an Implicit Reference by Reynolds[30].

The basic idea is to represent the Lvalue as a pair of functions. One of these, the load function, is a function of no arguments and it produces the value contained in the generalized Lvalue when it is used. The other function, the update function, is a function of one argument and when it is used it updates the value of generalized Lvalue with its argument. It should be pointed out that both functions may perform a large amount of computation to produce or store a value. For example, the update function might encode its argument before storing it into the internal Lvalue and the load function would decode it. This might be a way to save storage space.

A number of uses for a LUP are given in the paper by Reynolds[30]. However, several obvious S-PAL uses are given here for completeness. One very good use for LUPs would be to implement the idea of shaped locs. Although the set containment problem would not be solved, it is possible to build a verifier into the update function. The update would only be completed if the object being assigned satisfied the verifier.

The LUP also allows the implementation of the SUBSTR pseudovariable of PL/I. This allows assignment to an internal segment of a string without affecting the surrounding part of the string. It is a character for character replacement operation. This could be implemented in S-PAL by a function of three arguments, which, when applied to an Lvalue holding a string (a tuple of characters) and two integers delimiting the segment to be replaced, would produce an LUP. When the update function of this LUP is invoked, it would check to make sure the segment was of the correct size and would compute a new string with its argument replacing the old segment and would assign that to the Lvalue. If the string was a tuple of locs of characters, then the update function would not need to compute a new string but could instead replace each character of the segment of the old string with the corresponding character of its argument.

Computing Descriptors

It should be possible to give several different structures to the same set of data objects. This is useful when some subroutine a user wishes to use requires a slightly different format than the one in which the data is currently stored. If this alternative format is not too different from the existing format it should be possible to define the alternative structure on the same data. For example one might want to define a tuple which is composed of the even indexed components of another tuple. This implies multiplying every index for the new tuple by two to get the old tuple index. This type of alternate description is like that found in the DEFINED attribute of PL/I and the REDEFINES verb of COBOL.

In some cases the new description will be built on the original data and in other cases the new description will be phrased in terms of the existing structure. In the latter case, it is possible to build some alternate descriptions by embedding the original data function in a new function which maps its selectors into the selector set of the original function.

This could be done in the tuple example above. Further research is needed to decide if this will always suffice and how much efficiency is lost this way.

There is one other way to compute new descriptions or structures. This is the method used by Standish[33]. He provided modifiers which customized existing structures for particular uses. He also provided operators for combining several different structures into a single structure.

Syntactic Conveniences

There are several syntactic sugarings which might be considered for extensions. Two of these are trivial and one is more complex. One useful sugaring would be a facility for abbreviating long selector chains. One way to do this would be to give a name to a chain of selectors and to use the name instead of the chain. A second useful facility would be the ability to embed constants in a structure definition. They would be used to define components which never varied. That is, these components would always be filled in by the constructor and it would not be necessary to specify values for these components in the argument tuple.

The third sugaring is actually the most useful. It is often the case when large static structures are being defined that the various substructures are identical in format to previously defined structures. Therefore, it would be nice to be able to refer to those previous definitions to save copying the whole definition into the new structure. This function is provided by the LIKE attribute in PL/I and COBOL.

Basically, the idea is to copy the text of the previous definition into the place in the new definition. A textual copy is used so that the names of the constructors and predicates will be properly qualified for the new structure. This idea can be extended to provide modifiers, like those of Standish, which would make small modifications on the text as it is substituted into the new definition. One might be able to change the tag, the name of a selector, to fill in a constant value, etc.

Parameterized Definitions

There is one special case of the LIKE attribute which is worth separating out. This is the parameterized structure definition. This concept was used by both Standish [33] and Reynolds[31]. It is used for structure definitions which define a set of different data structure with very much the same description. For example, the set of **n x m** matrices forms a parameterized set of data structures where the parameters are the number of rows and the number of columns. It should be possible to write one definition for a matrix and to fill in the bounds at construction time.

It is important to note that this is not the same as a tuple which can vary in size. The tuple may be augmented at any time. Each member of a parameterized set has its parameters fixed when it is constructed and they may not vary after that. The values of these parameters complete the type information for the data structure. Because the parameter values are often needed when the data structure is processed, Reynolds provides dummy variables positions in his type checking predicates. These dummy variables are set as part of the type verification. They can then be used in the processing algorithm. This saves an extra reference to the data structure to find the bounds after the structure is verified.

Mixed Domain Data Functions

There is one feature of S-PAL which does not seem to be worth the complications it introduces into the formal definition. This feature is the capability of mixing symbolic and integer selectors. Most languages do not provide this feature. One reason might be that it is very easy to get almost the same effect by inserting an extra level in the structure at the point where the tuple part would begin. This extra level would contain the tuple part of the one level form. For example, the structure for chemical atoms given in Chapter IV (8) could be rewritten as

<u>def</u> ATOM <u>which</u> <u>has</u>

NAME <u>which</u> IsSTRING

<u>also</u> VALENCE <u>which</u> IsINTEGER

<u>also</u> BOND <u>which</u> IsTUPLE

Then if Carbon was an ATOM you would refer to the second bond by Carbon BOND 2 instead of Carbon 2 which would be used for the definition in Chapter IV. Because the extra level does not seem to be at all offensive, it is suggested that mixed domain functions not be allowed.

This, however, is not all there is to the problem. While the above example does not show it, it must be possible to have substructures below a tuple level. Therefore, the syntax for the tuple option must be modified to remove the symbolic alternatives and to allow structure definitions within the components of the tuple.

<u>Where will the Future Lead</u>

Almost all the languages which have a capability for structuring data have what might be called a middle level data structuring capability. It is not as low as the machine dependent bit oriented languages, but it is not quite at the level of some of the other features of higher level languages. They can be characterized as being node oriented and algorithmically connected. By this I mean that the user must allocate the nodes of the structure individually and construct a whole data base piece by piece.

This is still a relatively primitive facility. It should be possible within the near future to free the user from writing the algorithm which connects the nodes together. Instead, he should be able to specify (allocate) a set of nodes and for this set of nodes provide a list of all the connections the nodes should have. The machine would then make the connections given in the list in some optimal order and in parallel if possible.

This is only a first step. Many of the information management systems now in development go beyond this simple level. In these systems it is possible to specify data nodes and the relationships that these nodes should have to other nodes. The system then constructs a representation for those relationships and builds the data base with that representation.

The ultimate goal might be a system where the user specifies several sets of data, a set of attributes possessed by that data, and a set of constraints or relations between the data items. The system would take this information and build a data base where the constraints were satisfied. It is easy to visualize all kinds of problems with this approach. For example, when does a set of constraints have a solution? When is the solution unique? There is still much work to be dcne in the field of data structures.

Appendix A. The Complete S-PAL Syntax

Abbreviated S-PAL Syntax

```
P    ::=  { def D }_1^∞   |  E

E    ::=  let D in E  |  fn V . E  |  E1
E1   ::=  E2 where D2  |  E2
E2   ::=  valof C  |  C

C    ::=  C1; C  |  C1

C1   ::=  { NAME : }_0^∞ C2

C2   ::=  test B ifso C2 ifnot C2  |  test B ifnot C2 ifso C2
          |  if B do C1  |  unless B do C1
          |  while B do C1  |  until B do C1  |  C3

C3   ::=  T := T  |  goto R  |  dummy  |  res T  |  T

T    ::=  T1  { , T1 }_0^∞
T1   ::=  T1 aug T2  |  T2
T2   ::=  T3 at T3  |  T3
T3   ::=  B -> T3 BAR T3  |  B

B    ::=  B or B1  |  B1
B1   ::=  B1 & B2  |  B2
B2   ::=  not B3  |  B3
B3   ::=  A RL A  |  A

A    ::=  A + A1  |  A - A1  |  + A1  |  - A1  |  A1
A1   ::=  A1 * A2  |  A1 / A2  |  A2
A2   ::=  A3 ** A2  |  A3
A3   ::=  R  |  val R  |  loc R  |  A3 % NAME R

R    ::=  R2 of R  |  R1
R1   ::=  R1 R2 | R2
R2   ::=  NUMERIC  |  QUOTATION  |  TRUTHVALUE  |  NAME  |  nil
           |  ( E )  |  E

D    ::=  D1 within D  |  D1
D1   ::=  D2  { and D2 }_0^∞
D2   ::=  rec D3  |  D3
D3   ::=  NAME  { , NAME }_0^∞ = E  |  NAME V = E
           |  ( D )  |  D  |  S  |  P

V    ::=  V V1  |  V1
V1   ::=  NAME  |  ( NAME  { , NAME }_0^∞ )  |  ( )
```

RL ::= gr | ge | eq | ne | ls | le

S ::= ATOM which has S1

S1 ::= $\{$ S2 also $\}_1^\infty$ S2 | $\{$ S2 also $\}_0^\infty$ tuple | only S2

S2 ::= ATOM | (S) | P

P ::= ATOM which P1 | ATOM which P2

P1 ::= P3 $\{$ else P3 $\}_0^\infty$

P2 ::= is (S1) $\{$ else p3 $\}_1^\infty$

P3 ::= ATOM | is (S) | is (P)

Appendix B. The Gedanken Evaluator for S-PAL

```
//  Definitions Concerning Lists

    def  t x = x 1

    and  r x = x 2

    and  Push (x, s) = x, s

    def  2d x = t (r x)

    and  r2 x = r (r x)

    and  r3 x = r (r (r x))

    and rec  Prefix (x, y) =
                  Null y -> x
                | Null x -> y
                | [t x, Prefix (r x, y)]

    def  Tag n s = Aug s n

    and  Istag s n = n eq s (Order s)

    and  Sons = Order s  -  1

    and  Segment (x,i,j) = O(i,nil)
             where rec O(k,t) =
                 k gr j -> t | O(k+1,Aug t (x k))

//  Definitions Concerning λ-expressions

    def  bV x = x 2

    def  Body x = x 3

    and  Env x = x 4

    and  Isλexp x =
         Istuple x -> x 1 eq 'λ' | false

    and  Isλclosure = Isλexp

    and  Makeλclosure x y = Aug x y
```

```
def rec  Lookup (n, e) =
     n eq e 1 -> e 2 | Lookup (n, e 3)

def  Islabel x =
     Istuple x
      -> Order x eq 4
          -> x 4 eq 'Δ'
          | false
      | false

def  Tagof x = x (Order x)

def rec  Decompose (n, v, e) =
     test  Isvariable n
     ifso  n, v, e
     ifnot
       [Order v ne Order n -> error | Q 1 e
            where rec  Q k s =
                 k gr Order n -> s
               | Q (k+1) Decompose (n k, vk, s)]
```

// Definition of Makecontrol and subsidiary functions

// The subsidiary functions for structure definitions

```
def  MakeS (q, s) =
     Tag 'γ' (MakeStr, Tag 'τ' (q, s 1, s 2))

and  MakeP (q, p) =
     Tag 'γ' (IsStr, Tag 'τ' (q, p))

and  SimpleNS (q, s, p) =
     let  Lhs = (QualN 'Make' q, QualN 'Is' q)
     and  Ms = MakeS(q, s)
     and  Is = MackP(q, p) in
          [Lhs 2, Tag '=' (Lhs, Tag 'τ'  Ms, Is )]

and  SimpleNP (q, p) =
     let  Lhs = QualN 'Is' q
     and  Is = MakeP(q, p) in
          [Lhs, Tag '=' (Lhs,Is)]

def  NS (x, n, q, c) =
     c eq 4 -> (n, MakeP(q, nil), nil)
         | Buildpack (x, n, SimpleNS(q, x 1, nil))

and  NP (x, n, q, c) =
     c eq 4 -> (n, MakeP(q, x 1), nil)
         | Buildpack (x, n, SimpleNP(q, x 1))
```

```
and  NB (x, n, q, c) =
     c eq 4 -> (n, MakeP(q, x 1 2), nil)
          | Buildpack (x, n, SimpleNS(q, x 1 1, x 1 2))

and  NT (p) = Tag 'SubS' (true, p 1, nil)

and  Buildpack (x, n, s) = Tag 'SubS' (n, s 1, y)
     where y = Istag x 'SV' -> s 2 | AD(s 2, x 2)

def  Combine (a, d) =
     d eq nil -> Tag 'SV' (Aug nil a)
               | Tag 'SS' (a, AD d)

def rec  US (x, q, c) =
         let s,p,d = Split (x,q,c eq 4 -> 4 | 3)
            in Combine (Arg, d)
            where Arg = s (Order s) -> (s, p)
                        | (Aug s false, Aug p nil)

and  UP (x, q, c) = Combine (p, d)
     where s,p,d = Split(x, q, c eq 1 -> 2|c eq 3 -> 4|c)

and  AP (x, q, c) =
     lets,p,d = Split Segment(x,2,Order x),q,c eq 1 ->2|c
     and w = US (x 1, q, c)
        in  Combine [Tag 'Pair' (w 1, p), D1]
        where D1 = Istag w 'SS' -> Prefix(w 2, d) | d

and  Split (x, q, c) = Q (1,nil, nil, nil)
     where rec Q(k,s,p,d) =
          k gr Order x -> (Tag 'τ' s, Tag 'τ' p, d)
               |[ let m = Sub(x, q, c) in
                    Q(k+1, Aug s (m 1), Aug p (m 2), D1)
                    where  D1 = m 3 eq nil -> d | Aug d (m 3)]

and  Sub (x, q, c) =
     let  Type = Istag x in
       Type 'which has' -> NS[US(x 2, m, c),x 1, m, c]
     | Type 'which'     -> NP[UP(x 2, m, c),x 1, m, c]
     | Type 'is/has'    -> NB[AP(x 2, m, c),x 1, m, c]
     | Type 'tuple'     -> NT[UP(x 1, m, 4]
     | Type 'atom'      -> Tag 'SubS' [x, QualN 'Is' x, nil]
     | Tag 'Subs'[ nil, x, nil]
          where m = c ls 3 -> x 1 | QualN q (x 1)
```

```
//      The remaining definition standardizing functions

        def  WD u v =
             Tag '=' [v 1, Tag 'γ' (a, u 2)]
                   where  a = Tag 'λ' (u 1, v 2)

        and  RD w =
             Tag '=' [w 1, Tag 'γ', ('Y*', a)]
                   where  a = Tag 'λ' (w 1, w 2)

        and  FD u v = Q (Order u) v
             where rec  Q k s =
                    k eq 1 -> Tag '=' (u, s)
                            | Q (k-1) [Tag 'λ', (u k, s)]

        and  AD w = Q 1 nil nil
             where rec Q k s t =
                    k gr Order w -> Tag '=' (s, Tag 'τ' t )
                            | Q (k+1) [Aug s (w k 1)] [Aug t (w k 2)]

        def rec  D x =
             let  Type = Istag x
             in
               Type '='        -> x
             | Type 'within' -> WD [D (x 1)] [D (x 2)]
             | Type 'rec'    -> RD [D (x 1)]
             | Type 'ff'     -> FD (x 1) (x 2)
             | Type 'and'    -> AD (Q 1 nil
                                 where rec Q k t =
                                       k eq Order x -> t
                                          | Q(k+1)  Aug t (D (x k)) )
             | Type 'which has' -> NS US(x 2,x 1,1),x 1,x 1,1   3
             | Type 'which'     -> NP UP(x 2,nil,1),x 1,x 1,1   3
             | Type 'us/has'    -> NB AP(x 2,x 1,1),x 1,x 1,1   3
             | error
```

```
def rec  S x =
    test  Isidentifier x
    ifso  x
    ifnot
     let  Type = Istag x
     in
        Type '->'      -> Tag 'β' [S (x 1), S (x 2), S (x 3)]
      | Type 'test'    -> Tag 'β'[S(x 1), S(x 2), S (x 3)]
      | Type 'γ'       -> Tag 'γ'[S (x 1), S (x 2)]
      | Type 'λ'       -> Tag 'λ' [x 1, S (x 2)]
      | Type 'let'     -> Tag 'γ'[Tag 'λ' [w 1, S (x 2)], S (w 2)
                                 where  w = D (x 1)]
      | Type 'where'   -> Tag 'γ' [Tag 'λ' [w 1, S (x 1)], S (w 2)
                                 where  w = D (x 2)]
      | Type 'τ'       -> (let  n = Sons x
                           in
                           Q 1 nil
                            where rec  Q k t =
                             k gr n -> t
                            | Q (k+1)  Tag 'γ' (Tag 'γ'
                                         [Aug, t], S [x k]))
      | Type 'aug'     -> Tag 'aug' [S(x 1), S(x 2)]
      | Type '$'       -> Tag '$' [nil aug S(x 1)]
      | Type ';'       -> Tag ';' [S(x 1), S(x 2)]
      | Type ':='      ->
        {test Istag (x 1) 'τ'
         ifnot  Tag ':=' [S(x 1), S(x 2)]
         ifso   Tag 'γ' [Tag 'γ' ['Assign**', S(x 1)], S(x 2)]
        }
      | Type 'if'      -> Tag 'β' [S(x 1), S(x 2), 'dummy']
      | Type 'while' -> Tag 'ω' [S (x 1), S (x 2)]
      | Type 'until' -> Tag 'ω' [w, S (x 2)
                              where w = Tag 'γ' ['not', S (x 1)]]
      | Type 'goto'    -> S (x 1), 'goto'
      | Type ':'       -> (let w = S (x 2)
                           in
                           Istag w 'Δ' -> Tag 'Δ' (w 1 aug x 1 aug w
                                                   w 2)
                                     |  Tag 'Δ' [(x 1,  ),
                                                 where    = w, '#'])
      | error

def  Combine (s, t) = Q 1 s
    where rec Q k w =
         k gr Order t -> w | Q (k+1) (w aug t k)
```

```
def rec  L x =
     test  Isidentifier x
     ifso x
     ifnot
       [let Type = Istag x
        in
          Type 'Δ' -> [let u, v = x 1, L (x 2)
                       in
                       test  Istag v 'Δ'
                       ifso  Tag 'Δ'  Combine[(u, v 1), v 2 ]
                       ifnot Tag 'Δ' (u, v)]
        | Type 'ω' -> [let u, v = L (x 1), L (x 2)
                       in
                       test  Istag v 'Δ'
                       ifso  Tag 'Δ' [v 1, Tag ' ' (u, v 2)]
                       ifnot Tag 'ω' (u, v)]
        | Type ';' -> (let u, v = L (x 1), L (x 2)
                       in
                       test  Istag u 'Δ'
                       ifso
                          test  Istag v 'Δ'
                          ifso  Tag 'Δ' [w, Tag ';' (u 2, v 2)
                                         where w = Combine (u 1, v 1
                          ifnot Tag 'Δ' [u 1, Tag ';' (u 2, v)]
                       ifnot
                          test  Istag v 'Δ'
                          ifso  Tag 'Δ' [v 1, Tag ';' (u, v 2)]
                          ifnot  Tag ';'[ u, v )]
        | Type 'β' -> [let w, u, v = L (x 1), L (x 2), L (x 3)
                       in
                       test  Istag u 'Δ'
                       ifso
                          test  Istag v 'Δ'
                          ifso  Tag 'Δ' [s, Tag 'β' (w, u 2, v 2)
                                           where s = Combine (u 1, v)]
                          ifnot Tag 'Δ' [u 1, Tag 'β' (w, u 2, v)]
                       ifnot
                          test  Istag v 'Δ'
                          ifso  Tag 'Δ' [v 1, Tag 'β' (w, u, v 2)]
                          ifnot  Tag 'β' (w, u, v)]
        | Type '#' -> [let u = L (x 1)
                       in
                       test  Istag u 'Δ'
                       ifso  [x := u 2, '#';
                                Tag ' ' (u 1, x)]
                       ifnot (x := u, '#';
                                x)
        | Type 'λ' -> Tag 'λ' [x 1, L (x 2)]
        | Sons x eq 2 -> L (x 1), L (x 2), Tagof x
        | Sons eq 1 -> L (x 1), Tagof x
        | error ]
```

```
def rec  F (x, c) =
    test  Isidentifier x
    ifso  Push (x, c)
    ifnot
     let  Type = Istag x
     in
       Type '#' -> [x := F (x 1, c) ; x]
     | Type 'Δ' -> Push ('Δ', Push[ x 1, Push (F [x 2, nil], c)].)
     | Type 'β' -> (let δ = F (x 2, c), F (x 3, c)
                   in
                     F[x 1, Push ('β', δ)])
     | Type 'ω' -> [let δ, s = nil, ('dummy', c)
                   in
                   let t = F [x 2, (';', δ)]
                   in
                   δ := F (x 1, ['β', (t, s)])]
     | Type 'λ' -> Push ['λ', (x 1, SubC), c
                        where SubC = F (x 2, nil)]
     | Type ';' -> F (x 1, Push [';', F (x 2, c)])
     | Sons x eq 2 -> F (x 2, F (x 1, Push (tagof x, c)))
     | Sons x eq 1 -> F [x 1, Push (Tagof x, c)]
     | error]

def  Makecontrol P = F[L (S P), nil]
def  Contents (Memory, Address) =
     Look (Memory 2)
          where rec Look Mem =
                Address eq Mem 1 -> Mem 2  //Found.
              | Look (Mem 3)               //Keep Looking

and Update (Memory, Address, Value) =
     Memory 1, (Address, Value, Memory 2)

and  Extend Memory =
     let Nextcell = 1 + Memory 1
     in
     let  NextMemory = Nextcell, (Nextcell, nil, Memory 2)
     in
     NextMemory, Nextcell

def  C, S, E, D, M = nil, nil, PE, M


def  Store x =
     let  m. a = Extend M
     in
     M := Update (m. a. x);
     a

def  Lval x =
     Isaddress x -> x | Store x
```

```
//    State Transformations

      def  Subprobexit () =
                   C, S, E, D := D 1, Push [t S, D 2], D 3, D 4, M

      def  Evalconstant () =
                   C, S := r C, Push [w, S]
                        where w = val (t C)

      def  Evalvariable (C, S, E, D) =
                   C, S := r C, Push [w, S]
                        where w = Lookup (t C, E)

      def  Evalλexp () =
                   C, S := r C, Push [Newλclosure, S]
                        where  Newλclosure = Makeλclosure (t C) E

      def  Evalconditional () =
           C, S := (t S -> 2d C | r2 C), r S

      def  Applybasic () =
                C, S := r C, Push [w, r2 S]
                where w = IsLfcn(t S) -> a#pply (t S) (2d S)
                              | apply (t S)  Rval[(M,2d S)]

      def  NewLval () =
           let  m, a = Extend M
           in
           S, M := Newstack, NewMem
                where (Newstack = Push [t S, Push (a, r2 S)]
                       and
                       NewMen = Update [m, a, 2d S )]

      def  Applyλclosure () =
                let Newenv = Decompose [bV (t S), 2d S, Env (t S)]
                and Newdump = r C, r2 S, E, D
                in
                C, S, E, D := Body (t S), nil, Newenv, Newdump

      def  Assign () =
           test   Isaddress (t S)
           ifnot  C, S := r C, Push (dummy, r2 S)
           ifso  [C, S, M := r C, Push (dummy, r2 S), NewMem
                    where NewMem = Update (M, t S, Rval (M, 2d S))]

      def  Popstack () =
           C, S := r C, r S
```

```
def  Extendtuple () =
          [C, S := r C, Push (Newtuple, r2 S)
               where  Newtuple = Aug (t S) (2d S)]

def  LtoR() =
     S := Push [Contents (M, t S), 2d S]

def  Stepcontrol () =
     C := r C

def  Makelabels ( ) =
     let  δ, P = $E, 2d C
     and  Newdump = r3 C, $S, $E, $D
     and  j, k = 1, Order (2d C)
     in
     while  j le k do
       (let Labelval = P(j+1), δ, Newdump, 'Δ'
        in
        δ := P j, Labelval, $ δ;
        j := j+2
       );
     C, S, E, D := t(r2 C), nil, δ, Newdump


//   Main Programs

def  Transform () =
     test  Null C
     ifso  Subprobexit ()
     ifnot
       (  let  x = t C
          in
               Isconstant x      -> Evalconstant ()
           |   Isvariable x      -> Evalvariable ()
           |   Is exp x          -> Evalλexp ()
           |   x eq ';'          -> Popstack ()
           |   x eq ':='         -> Assign ()
           |   Isaddress (t S)   -> LtoR ()
           |   x eq 'β'          -> Evalconditional ()
           |   x eq 'val'        -> Stepcontrol ()
           |   x eq 'loc'        -> NewLval ()
           |   x eq 'aug'        -> Extendtuple ()
           |   x eq 'γ'
                 ->    Isλclosure (t S)
                       -> ()pplyλclosure ()
                       |  Applybasic ()
           |   Islabelval x      -> Makelabels()
           |   error
```

Appendix C. The Standardizing Functions for Typeless S-PAL

```
def rec D x =
    let  Type = Istag x
    in
      Type '='          -> x
    | Type 'within' -> WD [D (x 1)] [D (x 2)]
    | Type 'rec'        -> RD [D (x 1)]
    | Type 'ff'         -> FD  (x 1) (x 2)
    | Type 'and'        -> AD  (Q 1 nil)
                  where rec Q k t =
                     k eq Order x -> t
                            | Q(k+1) [Aug t (D (x k))]
    | Type 'which has' -> NS[US(x 2,x 1), x 1, x 1] 3
    | Type 'which'      -> NP[UP(x 2,nil), x 1, x 1] 3
    | Type 'is/has'     -> NB[AP(x 2,x 1), x 1, x 1] 3
    | error

def  SimpleNS (m,x,p,l) =
     let  Ms = Tag 'γ' (MakeStr, Tag 'τ' (m,x))
     and  Is = Tag 'γ' (IsStr, Tag 'τ' (m,p))
       in
         [Tag '='  l, Tag ' ' (Ms,Is)]

and  SimpleNP (m, x, l) =
     let  Is = Tag 'γ' (IsStr, Tag 'τ' (m,x))
       in
         [Tag '='  l,Is]

def  NS (x, n, m) =
     let l = [QualN 'Make' m, QualN 'Is' m]
       in   Istag x 'SS -> (n,l 2,AD[SimpleNS(m,x 1,nil,l),x 2])
                         | (n,l 2,SimpleNS(m, x, l, nil)

and  NP (x, n, m) =
     let l =[QualN 'Is' m]
       in   Istag x 'SS' -> (n,l,AD[SimpleNP(m,x 1,l),x 2])
                          | (n, l, SimpleNP(m, x, l))

and  NB (x, n, m) =
     let l = [QualN 'Make' m, QualN 'Is' m]
       in Istag x 'SS' -> (n,l 2,AD[SimpleNS(m,x 1 1,x 1 2,l),x 2])
                        | (n,l 2, SimpleNS(m,x 1,x 2,l))

def  Segment (x, i, j) = Q (i, nil)
     where rec Q (k, t) =
            k gr j -> t| Q (k+1, Aug t (x k))

and  Combine (a, d) =
          d eq nil -> a | Tag 'SS' (a, AD d)
```

```
def rec  Sub (x, q) =
     let Type = Istag x
     and m = QualN q (x 1)
     in
        Type 'which has' -> Tag 'SubS' NS[US(x 2,m),x 1,m]
     | Type 'which'      -> Tag 'SubS' NP[UP(x 2,q),x 1,m]
     | Type 'is/has'     -> Tag 'SubS' NB[AP(x 2,m),x 1,m]
     | Type 'tuple'      -> Tag 'SubS' [true,nil,nil]
     | Type 'atom'       -> Tag 'SubS' [x,QualN 'Is' x,nil]
     | Tag 'SubS' [nil,x,nil]

and  US (x, q) =
     let s,p,d = Split (x, q)
      in Combine (Tag 'τ' a, d)
         where a = s(Order s) -> s | Aug s false

and  UP (x, q) =
          Combine (Tag 'τ' p, d)
          where s,p,d = Split (x, q)

and  AP (x, q) =
     let s,p,d = Split(Segment(x,2,Order x), q)
     and w = US(x 1, q)
      in  Istag w 'SS' -> Combine[Pr(w 1,p),Prefix(w 2,d)]
                       | Combine[Pr(w, p), d]
          where Pr y = Tag 'pair' y

and  Split (x, q) = Q (1,nil,nil,nil)
     where rec Q (k,s,p,d) =
            k eq Order x -> s,p,d
                  | [let m = Sub(s k, q) in
                     Q (k+1,Aug s (m 1),Aug p (m 2),D1)
                     where D1 = m 3 eq nil -> d|Aug d (m 3)]
```

Appendix D. The Representation of S-PAL Data Functions

```
def  Decode (y,Sel) = D (Order Sel)
       where rec  D k =
          k eq 0 -> 0 | y eq (Sel k) -> k | D (k-1)

def  Buildset (Sel,t) = R(Order t - Order Sel + 1)
       where rec [ R k = k eq 0 -> Q(Order Sel - 1)
                                   | Aug [R(k-1)] k
             and  Q m = m eq 0 -> nil
                                  | Aug [Q(m-1)] [Sel m] ]

def  Buildvec (n,v) = S (1,nil)
       where rec S(m,t) = m eq n -> t | S[k+1,Aug t (loc v)]

def  Cstep1 (u,Sel) =
       let Chk = Buildvec (Order u, 0)
       and Nam = Buildvec (Order Sel-1,nil)
          in [Chk,Nam,Q (1,nil),u]
          where rec  Q (k,Un) =
             k gr Order u -> Un
                | Istag (u k) 'nqv' -> Q[k+1,Sort(Un,k)]
                                    | Q[k+1,Aug Un k]
             where  Sort (Unn,m) =
                [ let n = Decode(u m 2,Sel) in
                     n eq 0 or Chk n eq 1 -> undef
                          | (Chk n := 1 ; Nam n := m ; Unn)]
```

```
def  Cstep2 (Chk,Nam,Un,u) = R (1,1,nil
        where rec R(i,j,t) =
           i eq Order Chk -> t
                  Chk i eq 0 -> R(i+1,j+1,Aug t [u (Un j)])
                             | R(i+1,j,Aug t [u (Nam i) 1])

def  Canonical (u,Sel) = Cstep2(Cstep1(u,Sel))

def  Verify (V,t) = Q (1,true)
        where rec [ Q (k,Tv) =
                   k ge Order V -> Null (V k) -> Tv
                                             | R(k,Tv,V k)
                               | Q(k+1,Tv & V k (t k))
              and  R (m,Tv,Vr) = m gt Order t -> Tv
                                 | R(m+1,Tv & Vr (t m),Vr)

def  MakeStr (Tag,Sel,Ver) =
        let  n = Order Sel - 2
            in  Constructor
            where rec Constructor (u) =
               not Verify(Ver,t) -> undef
                 | fn y. IsATOM y ->
                                  y eq tag -> Tag
                                  | y eq domain -> Buildset(Sel,t)
                                  | y eq constructor ->Constructor
                                  | [ let k = Decode(y,Sel) in
                                          k eq 0 -> undef | t k]
                         | Sel(Order Sel) -> t(n+y) | undef
                  where  t = Canonical(u,Sel)
```

```
def  Test (Pred,v) = Q (Order Pred)
        where rec  Q k =
                    k eq 0 -> false | Q (k-1) or Pred k v


def  IsStr (Tag,Pred) =
        fn y. [Istuple Pred -> Test(Pred,y)
                             | Pred y]
              or y tag eq Tag
```

REFERENCES

[1] Balzer, R.M., "Dataless Programming," RAND Corporation, Memorandum RM-5290-ARPA, Santa Monica, Calif. 1967.

[2] Barron, D. W., et al, "The Main Features of CPL,' Comp. J. vol. 6, no. 2, 1963.

[3] Burstall, R. M., "Semantics of Assignment," Machine Intelligence 2, American Elsevier, New York, N.Y. 1968.

[4] Burstall R. M., and Popplestone, R. J., "Pop-2 Reference Manual," Machine Intelligence 2, American Elsevier, New York, N.Y. 1968.

[5] Christensen, C., "An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language," Interactive Systems for Experimental Applied Mathematics Academic Press, New York, N.Y., 1968.

[6] Church, A., The Calculi of Lambda Conversion, Annals of Mathematics Studies, No. 6, Princeton, N. J., Princeton University Press, 1941.

[7] Earley, J., "Toward an Understanding of Data Structures, Unpublished notes, University of California, Berkley, 1969.

[8] Evans, A., "Pal - A Language Designed for Teaching Programming Linguistics," Proceedings ACM National Conference, 1968.

[9] Evans, A., PAL - A Reference Manual and Primer, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Mass., 1970.

[10] Hammer, M. M. and Jorrand, P., "The Formal Definition of BASEL Part 1: Introduction," Computer Associates, Inc., Report CA-6908-1512, Wakefield, Mass., 1969.

[11] Hoare, C. A. R., "Record Handling," Symbol Manipulation Languages, Techniques, North Holland, Amsterdam, 1968.

[12a] Jorrand, P., "The Formal Definition of BASEL - Part 2: Compiler" Computer Associates, Inc., Report CA-6908-1512, Wakefield, Mass., 1969.

[12b] Jorrand P., "The Formal Definition of BASEL - Part 3: Interpreter," Computer Associates, Inc., Report CA-6908-1513 Wakefield, Mass., 1969.

[13] Landin, P. J., "The Mechanical Evaluation of Expressions," Comp. J., vol. 6, no. 4, pp. 308-320.

[14] Landin, P. J., "A Correspondence Between ALGOL 60 and Church's Lambda Notation," Comm. ACM., vol. 8, no. 2, pp. 89-101, vol. 8, no. 3, pp. 158-165, 1965.

[15] Landin, P. J., "A Formal Description of ALGOL 60," Formal Language Description Languages for Computer Programming, North Holland, Amsterdam, 1966.

[16] Landin, P. J., "A λ-Calculus Approach," Advances in Programming and Non-Numerical Computation, Pergamon Press, New York, N.Y., 1966.

[17] Landin, P. J., "The Next 700 Programming Languages," Comm. ACM, Vol. 9, No. 3, 1966.

[18] Laski, J., "The Morphology of Prex - An Essay in Meta-algorithmics," Machine Intelligence 3, American Elsevier, New York, N.Y., 1968.

[19] Laurence, N., "A Compiler Language for Data Structures," Proceedings ACM National Conference, 1968.

[20] Lucas, P., et al, "Method and Notation for the Formal Definition of Programming Languages," IBM Vienna Laboratory, Report TR 25.087, Vienna, Austria, 1968.

[21] McCarthy, J., et al, LISP 1.5 Programmers Manual, Cambridge, Mass., 1962.

[22] McCarthy, J., "Towards a Mathematical Science of Computation," IFIP Munich Conference 1962, North Holland, Amsterdam, 1963.

[23] McCarthy, John, "Definition of New Data Types in ALGOL X," ALGOL Bulletin No. 18, The Hague, The Netherlands, pp. 45-46, 1964.

[24] McCarthy, J., "A Formal Description of a Subset of ALGOL" Formal Language Description Languages for Computer Programming, North Holland, Amsterdam, 1966.

[25] Morris, J. H., "Lambda-Calculus Models of Programming Languages," Ph.D. Thesis, MAC-TR-57, Project MAC, Massachusetts Institute of Technology, Cambridge, Mass. 1968.

[26] Park, D., "Some Semantics for Data Structures," Machine Intelligence 3, American Elsevier, New York, N.Y., 1968.

[27] PL/I Language Specifications, IBM, Form Y33-6003-1, Mechanicsburg, Penna., 1969.

[28] Popplestone, R. J., "The Design Philosophy of POP-2, Machine Intelligence 3, American Elsevier, New York, N.Y., 1968.

[29] Richards, M., "BCPL: A Tool for Compiler Writing and Systems Programming," Spring Joint Computer Conference 1969, AFIPS Press, Montvale, New Jersey, 1969.

[30a] Reynolds, J. C., "GEDANKEN - A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept," CACM, vol. 13, no. 5, pp. 308-319, 1970.

[30b] Reynolds, J. C., "GEDANKEN - A Simple Typeless Language which Permits Functional Data Structures and Coroutines," ANL-7621, Argonne Nat. Lab., Argonne, Ill., 1969.

[31] Reynolds, J. C., "A Set Theoretic Approach to the Concept of Type," Nato Conference, Rome Italy, 1969.

[32] Ross, Douglas T., "A Generalized Technique for Symbol Manipulation and Numerical Calculation, Comm. ACM, vol. 4, no. 3, pp. 147-150, 1961.

[33] Standish, T. A., "A Data Definition Facility for Programming Languages," Ph.D. Thesis, Carnegie Institute of Technology, Pittsburgh, Penna., 1967.

[34] Strachey, C., "Towards a Formal Semantics," Formal Language Description Languages for Computer Programming, North Holland, Amsterdam, 1966.

[35] Strachey, C., "Fundamental Concepts in Programming Languages," NATO Conference, Copenhagen, 1967.

[36] USA Standard COBOL, United States of America Standards Institute, Report X3.23-1968, New York, N.Y., 1969.

[37] van Wijngaarden, A., et al, Report on the Algorithmic Language ALGOL 68, Mathematish Centrum, Report MR101, Amsterdam, 1969.

[38] Vigor, D. B., "Data Representation - The Key to Conceptualisation," Machine Intelligence 2, American Elsevier, New York, N.Y., 1968.

[39] Woodger, M., "The Description of Computing Processes: Some Observations on Automatic Programming and ALGOL 60," Annual Review in Automatic Programming, vol. 3, Pergamon Press, New York, N.Y., 1963.

[40] Wozencraft, J. M., and Evans, A., Notes on Programming Linguistics, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Mass., 1969.