position is that it permits the definition of new functional forms, in effect, merely by defining new functions. It also permits one to write recursive functions without a definition.

We give one more example of a controlling function for a functional form: Def pCONS -= otapplyotlodistr. This definition results in <CONS,fi ..... fn>--where the f~ are objects--representing the same function as [pfl ..... pfn]. The following shows this.

(p<CONS,fi ..... fn>):X

= (#CONS):<<CONS, fi ..... fn >,X>

by metacomposition

= aapplyotlodistr:<<CONS,fi ..... fn>,X>

by def of pCONS

= aapply:<<f~,x> ..... <fn,X>>

by def of tl and distr and o

= <apply:<fi,x> ..... apply:<fn, X>>

by def of a

= <(fx:x) ..... (fn:X)>

by def of apply.

In evaluating the last expression, the meaning function will produce the meaning of each application, giving pJ~:x as the ith element.

Usually, in describing the function represented by a sequence, we shall give its overall effect rather than show how its controlling operator achieves that effect. Thus we would simply write

(p<CONS, ffi ..... f~>):x = <(ffi:x) ..... (f~:x)>

instead of the more detailed account above.

We need a controlling operator, COMP, to give us

sequences representing the functional form composition.

We take pCOMP to be a primitive function such that,

for all objects x,

(p<COMe,fl ..... fn>):x

= (fi:(f2:(... :(f~:x)...))) for n _> 1.

(I am indebted to Paul Me Jones for his observation that

ordinary composition could be achieved by this primitive

function rather than by using two composition rules in

the basic semantics, as was done in an earlier paper

[2].)

Although FFP systems permit the definition and

investigation of new functional forms, it is to be expected

that most programming would use a fixed set of forms

(whose controlling operators are primitives), as in FP, so

that the algebraic laws for those forms could be em-

ployed, and so that a structured programming style could

be used based on those forms.

In addition to its use in defining functional forms,

metacomposition can be used to create recursive func-

tions directly without the use of recursive definitions of

the form Deff ~ E(f). For example, if pMLAST

nullotlo2 ~

lo2; applyo[1, tlo2], then p<MLAST>

-=

last, where last:x m x = <xl ..... Xn> ~ X~; &. Thus the

operator <MLAST>

works as follows:

#(<MLAST>:<A,B>)

633

= #(pMLAST:<<MLAST>,<A,B>>)

by metacomposition

= #(applyo[1, tlo2]:<<MLAST>,<A,B>>)

= ~t(apply:<<MLAST>,<B>>)

= #(<MLAST>:<B>)

= ix(pMLAST:<<MLAST>,<B>>)

= #(lo2:<<MLAST>,<B>>)

=B.

13.3.3 Summary of the properties of p and #. So far

we have shown how p maps atoms and sequences into

functions and how those functions map objects into

expressions. Actually, p and all FFP functions can be

extended so that they are defmed for all expressions.

With such extensions the properties of p and/~ can be

summarized as follows:

1) # E [expressions -* objects].

2) If x is an object, #x = x.

3) If e is an expression and e = <el ..... en>, then

#e = <#el, ..., #en>.

4) p E [expressions ~ [expressions ~ expressions]].

5) For any expression e, pe = p~e).

6) If x is an object and e an expression, then

ox:e = px:(ge).

7) If x and y are objects, then #(x:y) = #(Ox:y). In words: the meaning of an FFP application (x:y) is found by applying px, the function represented by x, to y and then finding the meaning of the resulting expression (which is usually an object and is then its own meaning).

13.3.4 Cells, fetching, and storing. For a number of reasons it is convenient to create functions which serve as names. In particular, we shall need this facility in describing the semantics of det'mitions in FFP systems. To introduce naming functions, that is, the ability to fetch the contents of a cell with a given name from a store (a sequence of cells) and to store a cell with given name and contents in such a sequence, we introduce objects called cells and two new functional forms, fetch and store.

Cells

A cell is a triple <CELL, name, contents>. We use this form instead of the pair <name, contents> so that cells can be distinguished from ordinary pairs.

Fetch

The functional form fetch takes an object n as its parameter (n is customarily an atom serving as a name);

it is written I'n (read "fetch n"). Its definition for objects

n and x is

I"n:x -= x = ~ ~ #; atom:x ~ ±;

(l:x) = <CELL,n,c> ~ c; ~'notl:x,

where # is the atom "default." Thus I'n (fetch n) applied to a sequence gives the contents of the first cell in the sequence whose name is n; If there is no cell named n, the result is default, #. Thus I'n is the name function for the name n. (We assume that pFETCH is the primitive function such that p<FETCH, n> ~ I"n. Note that ~n simply passes over elements in its operand that are not cells.)



position is that it permits the definition of new functional forms, in effect, merely by defining new functions. It also permits one to write recursive functions without a definition.

We give one more example of a controlling function for a functional form: **Def** $\rho CONS \equiv \alpha$apply∘tl∘distr. This definition results in $<CONS, f_1, \ldots, f_n>$—where the $f_i$ are objects—representing the same function as $[\rho f_1, \ldots, \rho f_n]$. The following shows this.

$$(\rho<CONS, f_1, \ldots, f_n>):x$$
$$= (\rho CONS):<<CONS, f_1, \ldots, f_n>, x> \quad \text{by metacomposition}$$
$$= \alpha\text{apply}\circ\text{tl}\circ\text{distr}:<<CONS, f_1, \ldots, f_n>, x> \quad \text{by def of } \rho CONS$$

$= \mu(\rho MLAST:<<MLAST>$

$= \mu(\text{apply}\circ[1, \text{tl}\circ 2]:<<MLAST>,$
$= \mu(\text{apply}:<<MLAST>,<B>>)$
$= \mu(<MLAST>:<B>)$
$= \mu(\rho MLAST:<<MLAST>,<B>$
$= \mu(1\circ 2:<<MLAST>,<B>>)$
$= B.$

**13.3.3 Summary of the propertie** we have shown how $\rho$ maps atoms functions and how those function expressions. Actually, $\rho$ and all FF extended so that they are defined With such extensions the properties summarized as follows: