Daisy 1.0 Reference Manual

by

Anne T. Kohlstaedt

Computer Science Department

Indiana University

Bloomington, Indiana 47405

TECHNICAL REPORT NO. 119

DAISY 1.0 REFERENCE MANUAL

ANNE T. KOHLSTAEDT

NOVEMBER 19, 1981

This work was supported by NSF Grant MCS77-22325.

Indiana University Applicative Programming for Systems Project (APS)
APS Memo 2

Daisy 1.0 Reference Manual

by

Anne T. Kohlstaedt

Revision 5

Abstract: This manual describes the PDP-10 Daisy 1.0 system developed by the IU APS Project. It is intended as a reference manual to help programmers already familiar with the Project's work to use the Daisy 1.0 implementation.

This work was supported by NSF Grant MCS77-22325.

Table of Contents

CHAPTER 1

INTRODUCTION

This is intended to be a user manual for the 1.0 interpreter of the language Daisy. Daisy was designed and implemented by the Indiana University Applicative Programming for Systems Project. Daisy is based on the "applicative premise", that is, that an applicative language can be designed that is not only sufficient for all the programming one would ever want to do, but moreover is the language of choice for this programming. Daisy is a dynamic language. At any point in time she reflects the current state of our thinking. The 1.0 interpreter is a snapshot of Daisy circa 1980.

If you know LISP, learning Daisy should not be difficult, since Daisy is a LISP descendant. If LISP is unfamiliar, then you should probably get comfortable with LISP first and then come back to this manual to learn Daisy.

Daisy has her roots in some early work by Friedman and Wise on the ramifications of modifying the semantics of LISP's cons. Since then, another constructor (frons) has been added and the syntax changed considerably. If you read the body of literature that has been produced along the way (the bibliography also serves as a reading list), you'll get a good feel for why we like the language that we've implemented,

what it buys you in terms of ease in programming, possible efficiency on a multiprocessor, etc.. In these papers, too, you'll find some excellent programming examples. I recommend reading through the papers, using this manual to translate programs into the syntax currently in vogue, and then trying out the translated examples.

1.1 NOTATION IN THIS MANUAL

==> means "evaluates to"

= means "is the same thing as"

id means a particular instance of "id". Either "id" is a system function or else it has been specifically referred to in a previous example. Function names are not actually underlined when you run Daisy.

1.2 MISCELLANEOUS DAISY INFORMATION

You need an upper/lower-case terminal to run Daisy since Daisy's system function names are all lower-case identifiers.

1.3 RETURNABLE SYSTEM ATOMS

There are several special Daisy atoms that are returned as the result of certain evaluations, but that don't evaluate to themselves as in LISP.

|?| |?| is returned as the result of an error- producing computation.

true true is returned as the value of some functions. Unless you bind true to some value though (like, say, true), true ==> |?|

beta beta is returned as the value of a lambda expression.

1.4 WHERE TO ADDRESS COMMENTS

Your critical comments will be carefully considered in shaping future editions of this manual. They should be directed to

Anne Kohlstaedt
Computer Science Dept.
101 Lindley Hall
Indiana University
Bloomington, IN 47405

CHAPTER 2

SYSTEM USE

2.1 LOGGING INTO AND OUT OF DAISY

The following dialog is an annotated Daisy session. In it, a system user logs into Daisy, evaluates three forms (T, which is unbound, <1 2 3> which evaluates to (1 2 3), and evlst:parse:dski:@A'.1 which reads in a file), and then logs off. Characters that would have been typed by the user are underlined. Control characters are preceded by a ^.

.RU Daisy[50105,5004]

TRC:II,000000000000 — A systems programmer can open an output file here and turn on various internal traces. These are probably useless to anyone not involved in debugging the interpreter itself. Type a carriage return to default to "no output file, no trace".

LIMIT: 7604 — Daisy runs with a maximum of 7604 fern cells. You can enter another (smaller) memory size here or type a carriage return

to default to 7604.

' ' is the prompt character, though this may change from terminal to terminal. The user has now logged into Daisy.

'T The user types a T and hits RETURN.

(!?! Notice the (. Since Daisy's output is considered to be a list of characters, when output first begins to stream from the terminal it is preceded by a (.

!?! represents 'error', the result of an error producing computation. In this case, !?! is returned because T is unbound.

'<1 2 3> The user types in <1 2 3> and the system returns (1 2 3), the list in which each

(1 2 3) element of the specification has been evaluated (see Section 4.1).

'evlst:parse:dski:@A'.1 The user types a form that causes the file A.1 to be read in and evaluated. dski turns the file into a list of characters, parse parses the list into a list of forms, and evlst evaluates each form in turn,

returning a list of the results.

(world
interpolate) The file, A.1, contained two function definitions - one called world and one called interpolate.

'^Z To log off Daisy and return to the host operating system, type the end-of-file character, control-Z.

) The system input function (see consol, Chapter 8) converges to nil, ultimately closing off the initial (of Daisy's output stream.

DSI EXIT. The user has now logged out of Daisy.

2.2 CONTROL CHARACTERS

^Z Console input stream terminator / end of file character

^D Detach character. When followed by a carriage return, it allows a call of Daisy's console input function to detach without converging to a value. This permits instantiations of consol:prompt within multisets to simulate proper behavior with the user's help. (We would like to be able to periodically strobe the keyboard to determine whether a key

has been depressed and let consol to detach if not, but we are unable to do so because of Fortran's I/O interface.) For further details see consol, Chapter 8.

^C,^T, etc. As per the user's operating system.

2.3 COMMENTS

Comment lines are preceded by a | and terminated by a carriage return.

```
Ex:     | This is a comment line
        | So is this

        <A B    | Comments don't have to begin a line
         D E>   | or terminate a form.
```

CHAPTER 3

ATOMS

Atoms in Daisy, as in LISP, are indivisible chunks of information. There are two categories, identifiers and numbers.

3.1 IDENTIFIERS

An identifier is a character string, taken as a unit, that serves as the name of some Daisy form. Normally, an identifier must begin with a letter (upper or lower case) or one of the characters ", $, ^, or `. It may contain in addition to letters and digits, the characters ", #, $, %, +, -, /, ?, @, \, ^, and `.

NOTE

> While " may be typed as part of an identifier name, it causes Daisy's character input routine to cease trimming blanks until another " is encountered. See consol, Chapter 8 for details.

If an identifier is desired which contains some other character, that character must be preceded by a '. (The ' does NOT become part of the identifier.) Thus, '4HAND is a way to type in the identifier 4HAND,

and POLKA'.'. a way to specify POLKA.. . The character ' might be used to handle characters like tab, linefeed, backspace, etc.. No thorough investigation has been made of just what characters are allowed through the operating system and implementation language's I/O interface, though, so experimentation on your part will be necessary.

BNF:

```
<identifier> ::= <start char> <follow char string> |
                 ' <any character> <follow char string>
<start char> ::= A - Z | a - z | " | $ | ^ | `
<follow char string> ::= <follow char> <follow char string> |
                         ' <any character> <follow char string> |
                         <follow char> | ' <any character>
<follow char> ::= <start char> | 0 - 9 | # | % | + | - | / |
                  ? | @ | \ | |
```

Ex: B4, after, $$PAYROLL$$, DIRECT@, cons, Y?, '10%

Identifiers are used to hold onto ("bind") other Daisy forms. Binding may be accomplished explicitly by means of the mechanisms described in Chapter 6 or implicitly by the pairing of formal parameters to actual arguments during function application. When an identifier evaluates, it returns its current (static) binding.

Ex: B4 ==> whatever B4 is bound to

after ==> whatever after is bound to

3.2 NUMBERS

Daisy's numbers are represented internally as rationals, although they may be typed in as integers or rationals (a rational number is the quotient of two integers, like 5/2). Numbers evaluate to themselves. No automatic reduction is done during evaluation although the results of certain arithmetic functions are returned in reduced form. Integer values are limited to the range [-134217728, +134217728].

BNF:

<number> ::= <integer> | <integer> / <integer>

<integer ::= <sign> <digits> | <digits>

<digits> ::= <digit> <digits> | <digit>

<sign> ::= + | -

Ex:

Input form		Output form
3	==>	3
3/2	==>	3/2
+3/2	==>	3/2
3/-134217728	==>	-3/134217728
-134217728/+134217728	==>	-134217728/134217728

CHAPTER 4

LISTS AND THEIR SPECIFICATION

4.1 LISTS

Lists and atoms comprise the Daisy data space. Atoms and the primitive list [], also called nil, (), and <>, make up the base-level data while lists are "constructed" from base-level objects and other lists. A list is an ordered sequence of elements and is written bracketed with parentheses (). Lists evaluate to themselves.

```
Ex: (dookie (zeleika goliath) bub)  ==>  (dookie (zeleika goliath) bub)
    (dog cat witchety-grub)         ==>  (dog cat witchety-grub)
    ()                              ==>  []
```

Lists with infinite homogenous tails may be defined by writing a "*" as the last element. Thus

```
(add *)
```

with one "explicit" element, is conceptually an infinite list of adds, (add add add ...). (Actually it is constructed as a cyclic list.) The list of four explicit elements

(add sub mpy div *)

is also infinitely long. It's first three elements are the atoms add, sub, and mpy, while its remaining infinite tail is the list

(div *)

Every list except [] has a first and a rest.

Ex:	list	first	rest
	(A)	A	()
	(1 *)	1	(1 *)
	(A B)	A	(B)

Probing functions are used to access the contents of a list. The two primitive probing functions, in terms of which all other such functions can be written, are first and rest (see Chapter 7.2). first, when applied to a list, returns the lists' first element if it has one. rest returns what would remain of the list if its first element were removed.

Often the rest of a list is another list, but it is possible to define a list whose rest is of arbitrary type. This is like LISP's dotted pair, but the character "!" is used rather than ".".

(A ! B)

is a list whose first is A and whose rest is B. The lists

(A ! (B ! (C ! ())))

(A B C ! ())

(A B C)

are identical. In typing input to Daisy, any of the three expressions may be used. On output, Daisy prints the representation with fewest "!"s.

When Daisy builds a list, it suspends the evaluation of the list's elements. That is, computation of an element's value is deferred until a probe attempts to access it. This allows functions that create and manipulate non-finite lists to run and return results [10]. Furthermore, accessing the rest of a list causes no evaluation of the first, and vice versa. Thus lists with elements whose evaluation might not terminate (called "divergent" elements) may be constructed and manipulated without the manipulating computation itself necessarily diverging (see Chapter 7.1).

4.2 LIST SPECIFICATIONS

There are two ways to define a list: by naming its elements explicitly, as in the examples of Chapter 4.1, or by writing a construction specification for the list, called a fern. There are two

types of ferns: sequences and multisets.

Ferns look like lists but are written with different bracketing characters that indicate how the list is to be constructed. Like lists, ferns admit the use of "!" to specify structure and "*" to specify extent. The value of a fern is a list whose content depends on the environment in which the fern specification is evaluated. In other words, fern elements are themselves specifications (although not necessarily of lists) and must be evaluated.

4.2.1 Sequences

A sequence is enclosed in angle brackets <> and specifies both the order and content of a list.

Ex: Suppose an environment in which A ==> dookie, B ==> zeleika, C ==> goliath, D ==> bub, and E ==> (zeleika goliath)

```
<A <B C> D>   ==>  (dookie (zeleika goliath) bub)
<A E ! D>     ==>  (dookie (zeleika goliath) ! bub)
<B *>         ==>  (zeleika *)
<>            ==>  []
```

4.2.2 Multisets

A multiset is enclosed in curly brackets {}, and specifies only the content of a list and not the order of its elements as well. Whenever a multiset is probed, all it's elements are allowed to evaluate concurrently ("coaxed"). When one returns a value ("converges"). That element is promoted to the head of the list (remember, a multiset specifies a list) and is thereby excluded from future consideration as a member of the rest of the list. Elements converging to |?| (error producing computations) are not promoted until all other elements have converged.

Ex: Suppose an environment in which A ==> Schroedinger's, B ==> cat, C ==> |?|, and D diverges, that is cannot produce a result

```
{A B}      ==>  (Schroedinger's cat)
                        or
                (cat Schroedinger's)
{A B C C}  ==>  (Schroedinger's cat |?| |?|)
                        or
                (cat Schroedinger's |?| |?|)
{A C D}    ==>  (Schroedinger's
{}         ==>  []
```

Multisets allow programmers to deal with "indeterminite" real time behavior in an applicative fashion [7]. Elements are devised that converge to values in response to some external event taking place, such as a key being depressed at a terminal; first and rest then serve as

polling functions, picking out elements as they converge (see Chapter 7.1).

Multisets can also be used to collect sub-computations whose order of convergence is not important. This allows programmers to specify their algorithms as weakly as possible, and avoid introducing the notion of sequentiality when it is not necessary.

4.3 MIXED STRUCTURES

Daisy permits the specification of structures which are a mixture of sequence and multiset.

Ex: Suppose an environment where A ==> a, B ==> b, C ==> c,
and L is bound to {A ! <B C>}

{A ! <B C>} ==> (a b c), (b a c), or (b c a)

Note that the expression <B C> specifies that B's value must precede C's value. If B diverges then C is not promoted.

CHAPTER 5

APPLICATIVE FORMS

Function application in Daisy is right associative and denoted by an infix colon.

Ex: consol:A — The function consol is to be applied to the argument A.

or:{A B C} — The function or is applied to the multiset {A B C}.

x:y:z — The function x is applied to y:z.

5.1 FUNCTIONAL COMBINATION

Functional combination [6] is a syntactic tool that makes it easy for a programmer to express recursions that accumulate multiple results. A functional combination is indicated by the presence of a list of function names in the "function" position (or a form that evaluates to a list of function names). A list of lists, called the parameter matrix, (or a form that evaluates to such a list) must occur in the argument position. The "columns" of the parameter matrix become argument lists

to respective elements of the list of functions. The result of the functional combination, then, is a list of the results of the individual applications. For example:

```
(add mult):
<< 1   3 >     =   <add:<1 0> mult:<3 4>>   ==>   (1 12)
 < 0   4 >>
```

By adopting the style used in the above example, with elements of argument rows aligned vertically beneath the respective elements of the function list, programs that are both clear and concise may be written. The special identifier # may be used as a place holder in writing a row and is ignored as an argument in interpreting a column of the parameter matrix:

```
(cons sigma):
<< 1    2 >   =  <cons:<1 (2)> sigma:<2 3 4>>  ==>  ((1 2) 9)
 < (2)  3 >
 < #    4 >>
```

The following example taken from [3] illustrates the use of functional combination in a more representative setting. This is a program for dealing a deck of cards to two players in a game like War. The function deal takes a list deck as its argument and returns a list of two lists as its result, each composed of alternate elements from deck. The recursive call to deal is embedded within the functional

combination, and evaluates to the final row of the parameter matrix, namely the rest:rest:deck dealt into two lists.

```
deal:deck = if:< empty?:deck       <[][]>
                empty?:rest:deck   <deck []>
                (  cons     cons ):
               << 1:deck   2:deck >
                deal:rest:rest:deck > >
```

where the functions if, 1, 2, empty?, cons, and rest are as defined in subsequent chapters.

Starred structures combined with "guillotine rules" are particularly useful in expressing functional combinations. Daisy's guillotine rules differ slightly from those described in [6] and [4] but serve the same purpose - they allow you to write a starred structure where you really mean "as long as necessary" rather than "infinitely long". For example,

```
(add * ):          (add add add):
<< 1  * >    =     << 1   1   1 >
 < 1 2 3 > >         < 1   2   3 >>
```

adds 1 to each element of the list <1 2 3>, returning

(2 3 4).

The implemented guillotine rules are complicated. They break down into two cases, depending on whether the function list is starred or not.

1. Function list not starred Parameter rows longer than the function list are truncated. Shorter rows are right filled with #s.

```
Ex:  (f1  f2  f3  f4):<          (f1  f2  f3  f4):<
     <a11 a12 *     >      =     <a11 a12 a12 a12>
     <a21 a22 a23 a24 a25>       <a21 a22 a23 a24>
     <a31>>                      <a31  #   #   # >>
```

2. Function list is starred

   a. There is a non-starred parameter row

      The guillotine rule is applied, based on the first non-starred row of the parameter matrix.

      i. There are more explicitly named elements (i.e the non-starred elements) in the function list than in the first non-starred parameter row (or the same number).

         The function list is truncated just before the star. Parameter rows longer than the truncated function list are cut off at the same length. Shorter rows (including

perhaps the first non-starred parameter row) are right filled with #s.

```
Ex:  (f1  f2  f3  *):<          (f1  f2  f3 ):<
     <a11 a12>          =       <a11 a12  # >
     <a21 a22 a23>              <a21 a22 a23>
     <a31 *>>                   <a31 a31 a31>>
```

ii. There are fewer explicit elements in the function list than in the first non-starred parameter row.

The function list and longer parameter rows are truncated to be the same length as the first non-starred parameter row. Shorter parameter rows are backfilled with #s.

```
Ex:  (f1  f2  *):<              (f1  f2  f2):<
     <a11 a12 a13>      =       <a11 a12 a13>
     <a21>                      <a21  #   # >
     <a31 *>>                   <a31 a31 a31>>
```

b. All rows of the parameter matrix are starred.

The result is starred.

```
Ex:   (f1  f2  *):

     <<a11 a12 *>     =   <f1:<a11 a21> f2:<a12 a21> *>
      <a21 *>>
```

CHAPTER 6

ENVIRONMENTS AND BINDINGS

The Daisy interpreter evaluates expressions read from the console device. In order to evaluate each expression, however, values must be assigned to the variables occurring in the expression. Environments serve to store this association or "binding" information. Certain identifiers have already been bound when you initially log into Daisy. (Conceptually, this is true although these bindings are not implemented in the same way as other bindings discussed in this chapter.) These include numbers (bound to themselves), the empty lists [], <>, () and {} (bound to []), and the primitive functions described in Chapters 7 - 13.

Daisy has two environment types, a "shallow" environment where most global function definitions are stored, and a "deep" environment where global non-function declarations, some global function definitions, and local function and non-function (let and rec) definitions are bound. When an identifier's binding is sought, the shallow environment is always searched first. Thus a shallow global functional binding for an identifier will hide subsequent local bindings to the same identifier.

6.1 LAMBDA EXPRESSIONS

Function definitions are bindings of lambda expressions to identifiers. A lambda expression in Daisy has the syntax:

```
<lambda expr> ::= \( <formal parm> . <body> )
<formal parm> ::= <identifier> | <list> (see Section 4.1)
<body>        ::= <any valid Daisy expression>
```

```
Ex: \(X . add:<2 X>)
    \((A L) . if:<empty?:L       []
                 same?:<A 1:L>  @true
                 MEMBER:<A rest:L>   > )
    \(N . if:<eq?:<N 0> 1 mpy:<N FACTORIAL:dcr:N> > )
```

Lambda expressions evaluate to closures, returning a reference to the identifier beta. A closure consists of the lambda expression and an environment. The closing environment may be the "current" environment, but when the lambda expression's body contains no free variables, as is usually the case, the current environment may be discarded and the lambda expression closed in a new, empty environment. The user must recognize the situation though. If a function body contains no free variables, the alternate colon syntax

```
\( <formal parm> : <body> )
```

is used to signal the fact. By not carrying environments around except

where necessary, a more efficient use of available space can be made. You should be careful, though, when using this feature. An intermediate function in a chain of invocations, defined to have no free variables, will cause inner functions to lose their bindings to free variables bound outside the scope of the intermediate function.

```
Ex:  \(LOST . \(X : \(Y . LOST):1 ):2 ):3
                this : prevents \(Y.LOST) from finding
                a binding for LOST since \(X:(Y.LOST):1)
                is closed in an empty environment.
```

6.2 DEFINITIONS AND DECLARATIONS

Definitions and declarations create "permanent" global bindings. As side-effect producers, they live at top level, philosophically just "outside" Daisy's clean inner functional world.

6.2.1 Declarations

Declarations bind identifiers to non-functional values in the deep global environment, permanently extending it. Two alternate syntaxes exist to handle problems of lookahead in parsing.

```
        <identifier> = <form>
    and <identifier> = <form>.
```

Ex: A = add:<5 6>

A = add:<5 6>.

NOTE

The . serves to indicate that a : is NOT to follow, the result being that the declaration is done a little sooner. Otherwise, the scanner must look at the next line to make sure that :<more form> isn't trailing along.

Ex: I = add

:<5 6>

6.2.2 Definitions

Definitions can create function bindings in either the shallow or deep global environment. Again there are several alternate syntaxes. To shallowly bind a function definition, write:

<identifier> : <formal parm> = <body>

or <identifier> : <formal parm> = <body>.

. here serves the same purpose as in a declaration.

```
Ex:  FACTORIAL:N = if:<eq?:<N 0> 1
                        mpy:<N FACTORIAL:dcr:N>>
     FACTORIAL:N = if:<eq?:<N 0> 1
                        mpy:<N FACTORIAL:dcr:N>>.
```

To place a functional binding in the deep global environment, bind a lambda expression to an identifier:

```
<identifier> = <lambda expr>
```

```
Ex:  FACTORIAL = \(N . if:<eq?:<N 0> 1
                           mpy:<N FACTORIAL:dcr:N>> )
```

A programmer can specify that a function carries no free variables either by adopting the

```
<identifier> = <lambda expression>
```

syntax and specifying a lambda expression with no free variables, or by writing

```
<identifier> : <formal parm> =: <body>
or <identifier> : <formal parm> =: <body>.
```

```
Ex:  ADD2 = \(N:add:<2 N>)
     ADD2:N =: add:<2 N>
     ADD2:N =: add:<2 N>.
```

6.3 LET AND REC

The functions let and rec establish new local bindings, extending the current environment long enough to execute the expressions specified in the let or rec.

6.3.1 Let

let allows the Daisy programmer to extend the current environment with local nonrecursive definitions and declarations and evaluate an expression in this extended environment. The syntax is

```
<let expr>     ::=  let:( <local var> <local def> <expr> )
<local var>    ::=  <identifier> | <list>
<local def>    ::=  <expr>
<expr>         ::=  <any valid Daisy expression>
```

A let expression is equivalent to

```
\( <local var> . <expr> ) : <local def>
```

The expression making up the local definition, <local def>, is evaluated in the current environment and bound to the corresponding local formal parameter structure, <local var>, to create a temporarily enriched environment in which the expression, <expr>, is allowed to execute. On completion, the value of the expression is returned as the value of the let, and the local bindings are undone.

```
Ex: let:( A 2 add:<A 10>)      - extend the current environment
                                 by binding 2 to A, and evaluate
                                 add:<A 10> in that environment.
                               = \(A . add:<A 10>):2
                               ==> 12

    let:( (A ! B) <2 3>        - extend the current environment
          add:<A first:B> )      by binding 2 to A, <3> to B, and
                                 evaluate add:<A first:B> in the
                                 extended environment.
                               = \((A ! B) . add:<A first:B> ):
                                 <2 3>
                               ==> 5
```

```
let:( (ADD2 ADD3)              - bind ADD2 and ADD3 to the
     <\(N.add:<2 N>)             respective lambda expressions,
      \(N.add:<3 N>)>            and evaluate the form
     <ADD2:5 ADD3:6> )           <ADD2:5 ADD3:6> in the
                                 resulting environment.
                               = \((ADD2 ADD3).<ADD2:5 ADD3:6>):
                                 <\(N.add:<2 N>) \(N.add:<3 N>)>
                              ==> (7 9)
```

6.3.2 Rec

rec is similar to let except that the local definitions and declarations are allowed to refer to themselves and each other.

```
Ex: rec:( FACTORIAL
          \( N . if:<eq?:<N 0> 1 mpy:<N FACTORIAL:dcr:N>> )
          FACTORIAL:10 )
        - this expression always returns 10 factorial
          i.e. 10*9*8*7*6*5*4*3*2*1.
```

```
RSFF:(R S) =: rec:( (Qhi Qlo)  < <1 ! (NAND *):<R Qlo>>
                                 <0 ! (NAND *):<S Qhi>> >
                               < /(x . x) * >:<Qhi Qlo> )
```

- RSFF models the behavior of a synchronous RS flip flop [13]. R and S represent lists of values that are presented to the R and S inputs, respectively. Qhi and Qlo represent the lists of values presented by the flip flop's outputs.

6.4 STRUCTURED PARAMETERS

Each Daisy function takes only ONE argument. Consider the function definition

```
MEMBER:(A L) = if:< empty?:L            []
                    same?:<A first:L>   @true
                    MEMBER:<A rest:L>       >
```

and the function call

```
MEMBER:< person  APPEND-ALL:<NAACP NOW NRA> >.
```

In each, there is one argument or parameter. The one parameter of the function definition is the list (A L). The one argument of the function call is the list <person APPEND-ALL:<NAACP NOW NRA>>. The structure of the function call's argument, though, matches the structure of the function definition's parameter. A binds to person and L binds to the

form APPEND-ALL:<NAACP NOW NRA> because they occupy corresponding positions in the argument and parameter structures. So when you write the formal parameter part of a function definition, you are specifying the macroscopic structure of that parameter.

In Daisy we carry this further and allow you to define a parameter's microscopic structure, that is the internal structure of its constituents. For example, an alternate definition of MEMBER

```
MEMBER:(A (LA ! LD)) = if:< same?:<A LA>  @true
                            empty?:LD      []
                            MEMBER:<A LD>      >
```

specifies that the second element in the argument structure of a function call to MEMBER should be a nonempty list, and that the first of that list is to be implicitly bound to LA and the rest of that list to LD. (Since Daisy is suspended, these implicit firsts and rests are not done unless an element is "needed" during execution of the body of the function.) Notice, though, that you've lost a way to talk about the list as a whole. In particular, you can no longer check to see if the list is empty. You can only check the status of its rest, LD.

The function let (or rec), however, deals with this problem, allowing you to specify the detailed structure of an argument without losing a handle on its larger structural components. These final definitions of MEMBER, then, give both a name for the list L and its major constituents, LA and LD

```
MEMBER:(A L)  =:  let:( (LA ! LD)  L
                        if:< empty?:L       []
                             same?:<A LA>  @true
                             MEMBER:<A LD>       > )

MEMBER:(A L)  =:  if:< empty?:L    []
                       let:( (LA ! LD)  L
                             if:< same?:<A LA>  @true
                                  MEMBER:<A LD>       > ) >
```

CHAPTER 7

SIMPLE FUNCTIONS ON LISTS AND FERNS

7.1 CONSTRUCTORS

cons:<element list>

cons is a system primitive that allocates and defines the contents of a new binary list cell. Initially, all elements of a consed list are in a state of suspended evaluation, awaiting outside probing by the function first to converge to an ultimate value [5]. For example, consider the following:

NATURALS:N = cons:<N NATURALS:inc:N>

where inc is a function that increments its numeric argument by 1.

The function NATURALS builds an infinite list of natural numbers, starting from the number N. In conventional LISP, a call to NATURALS say, NATURALS:1, will never converge since neither argument to cons is suspended, and evaluation of the second leads to immediate unbounded recursion. In Daisy, though, since cons is suspended, NATURALS:1 converges immediately. Only relentless probing by the system's "printer process" causes convergence of

each natural number in turn, driving the recursion only as far as the next (suspended) cons [11]. NATURALS can, in fact, be printed with constant memory space since old list cells are reclaimed by the garbage collector as the natural numbers that they contain are printed and the printer's pointer into the structure moves on [9].

The system's own cons is suspended, as well as the user's. This results in call-by-name semantics since argument structures are never built until their elements are accessed in a function body. Thus

```
first:rest:<forever:1  plus:<3 4>>
where  forever:X = forever:X
```

returns the answer 7. The fact that forever:1 diverges has no effect on the result.

There is a shorthand notation using "!" for defining lists that avoids the explicit use of cons:

```
(ONE TWO)      =  cons:<@ONE cons:<@TWO nil>>
(ONE ! MORE)   =  cons:<@ONE @MORE>
<ONE TWO>      =  cons:<ONE cons:<TWO nil>>
<ONE ! MORE>   =  cons:<ONE MORE>
```

where @ is equivalent to LISP's function quote.

frons:<element multiset>

frons [1,2,7] is the suspended constructor used to build multisets.

The content of a multiset is defined when built, but the order of elements within it remains unspecified until they are accessed by means of first and rest. As the multiset is probed, the first element to converge takes its place at the front. Gradually, as the multiset acquires order, it becomes a list whose contents and order are immutable.

Thus

first:{A B C}

causes enough evaluative effort to be distributed to A, B, and C to force at least one of them to converge (if possible). The convergent element (or one of them if several should converge simultaneously) is promoted to the first of the multiset {A B C} and is returned as the value of 1:{A B C}.

Like lists, there is a short notation for fronsing together multisets:

{ELEMENT ANOTHER} = frons:<ELEMENT frons:<ANOTHER nil>>

{ELEMENT ! MORE} = frons:<ELEMENT MORE>

Notice that this is NOT the same thing as frons:{ELEMENT MORE}. The first definition builds a multiset with one more element than MORE, whereas the latter builds one with just two elements.

In the following example, adapted from [7], a multiset of I/O sequences is specified and the sequences interleaved. Each I/O sequence is a list of characters read from a terminal.

```
{read:tty1  read:tty2  read:ttyx}
where  read:terminal   = cons:<readch:terminal
                              read:terminal>
       readch:terminal = a function that reads a character
                         from the specified terminal and
                         side-effects the terminal's buffer.
```

We would like to choose an I/O sequence in which a character has been read, snatch the character, and frons the remainder of the sequence back into the multiset. This requires that an invocation of read not converge until a character is ready. But, since cons is suspended, such will not be the case. cons may converge regardless of whether readch has successfully read a character or not. In order to obtain the desired behaviour from I/O sequences, the function strictify must be introduced.

```
strictify:<a b>  =  if:<a b b>
```

strictify returns the value b if the evaluation of a terminates. If a's evaluation does not terminate, neither will strictify. strictify may now be used to process each I/O sequence before it is placed in the multiset.

```
{streamify:read:tty1 streamify:read:tty2 streamify:read:ttyx}

where  streamify:seq
           = if:<empty?:seq  ()
                 strictify:<1:seq
                           cons:<1:seq
                                streamify:rest:seq>>>
  and  1:x  =  first:x
```

Now a merge function that flattens the multiset of streamified I/O sequences can be defined, effectively interleaving the sequences.

```
merge:M = if:<empty?:M    M
             empty?:1:M  merge:rest:M
             cons:<1:1:M  merge:frons:<rest:1:M  rest:M>>>
```

Further examples of the use of frons in writing systems style programs may be found in [1,7,13,15].

7.2 PROBING FUNCTIONS

first:L

first coerces and returns a reference to the 1st element of L. In the case of a list or sequence, L's order is defined at construction time and its predetermined first element is coerced

into existence on invocation of first. If L is a multiset, however, then its elements are all coaxed simultaneously until one converges [1,2,7]. The first one to converge is returned as the value of first:L and promoted to the head of L (so that subsequent probes return consistent values). !?!, however, is never promoted (see Chapter 4).

Ex: first:(1 |?| 3) ==> 1

first:<1 |?| 3> ==> 1

first:{1 |?| 3} ==> 1 or 3

1:L, 2:L, ..., 134217728:L

Placing an integer, N, in a functional position is a way of selecting the Nth element from L.

Ex: 1:(A B C) = first:(A B C) ==> A

2:(A B C) ==> B

3:<1 3 add:<3 4>> ==> 7

rest:L

rest returns L without its first element. If L is a list or sequence then no evaluation of any suspensions in a first field are driven [5]. Thus

rest:rest:< div:<1 0> add:<3 4> >

coerces neither div:<1 0> nor add:<3 4>.

In the case of a multiset L, its elements begin evaluation in parallel until one converges (unless some element is already convergent) [1,2,7]. That convergent element is promoted to be the head of L and a reference to the rest of the multiset is returned.

Ex: rest:(1 |?| 3) ==> (|?| 3)

rest:<1 |?| 3> ==> (|?| 3)

rest:{1 2 3} ==> (1 2), (2 3), (3 1), (1 3), (3 2) or (2 1)

rest:{1 |?| 3} ==> (3 |?|) or (1 |?|)

rest:{forever:1 add:<3 4>}

$\underline{=}$ {forever:1}

if forever:X = forever:X, since forever:1 is a computation that never converges to a value.

CHAPTER 8

FANCY LIST/FERN FUNCTIONS

NOTE

In the examples in this chapter, the characters <new line> and <blank> may occur for various reasons. A <blank> may be typed by a user, returned as a list element by Daisy, or used by the printer to separate other list elements. Similarly, a <new line> may delimit a line of user input, a line of Daisy output, or itself be an element in a list. To help the reader differentiate among the various aspects of each character, some additional representations of <blank> and <new line> will be introduced. The character "_" will stand for a blank that has been typed by the user or returned as a list element. " " will continue to represent blanks generated by the print routine for readability. Likewise, "^M" followed by an effective <new line> will represent a <new line> that has either been typed by the user or returned as a list element by Daisy, while an effective <new line> alone will represent one generated by Daisy's print routine.

consol:prompt

The function consol takes a prompt character as its argument. It sets up a suspended input channel from the programmer's console, writing the prompt on the console screen whenever it is ready to process a line of input data. consol returns a list of the characters read, suspending itself after each carriage return/line feed. When ^Z is typed, consol immediately converges to nil.

```
Ex:  consol:@'%  ==>  %When_in_the^M
                      (_ W h e n _ i n _ t h e ^M
                      %_course^M
                       c o u r s e ^M
                      %^Z
                      )
```

NOTE

When ^Z follows other characters on the same line, however, those characters are lost. Type ^Z on a separate line!

```
Ex:  consol:@'%  ==>  %When_in_the_course_^Z
                      ()
```

consol processes input lines in stages. First, lines longer than 72 characters are truncated (due to the Fortran I/O interface) and lines shorter than 72 are padded to the right with blanks. Then sequences of blanks are compressed to one blank.

NOTE

If a " has been read, consol is supposed to forego compression of blanks until another " is encountered. This feature has not been released yet, though.

Finally, lines ending with a blank have a carriage return inserted at the right. To continue a short input line on the next line

without picking up any untyped blanks or carriage returns, end the initial line with a comment symbol |.

```
Ex: consol:@'% ==> %all____("____"____gars___
                    (_ a l l _ ( " _ " _ g a r s _ ^M
                   %eat):cash^M
                    e a t ) : c a s h _ ^M
                   %^Z
                   )

    consol:@'% ==> %aaa|^M
                   %ccccc^M
                    (_ a a a c c c c c _ ^M
                   %^Z
                   )
```

Instantiations of functions like consol within multisets should allow you to write applicative programs whose behavior is governed by indeterminite real time events (see Chapter 7.1). The form

```
first:{ rest:rest:consol:@A  rest:rest:consol:@B }
```

should cause consol:@A and consol:@B to run in parallel, and return a stream of characters from the first input channel to converge. For example, if a user begins typing characters at channel B, while channel A remains quiescent, the form should converge to B's

stream. However, because of the I/O interface of Daisy's implementation language, it is impossible to simulate running consol:@A and consol:@B concurrently without some outside help from the user. To execute a Fortran READ is to cause the system to hang until a line of data becomes available. We designed consol so that typing

^D <carriage return>

in response to a consol input prompt, causes consol to detach without converging to a value. This permits other instantiations of consol to then run (conceptually in parallel).

dski:filename

dski:<filename>

dski:<filename project#/programmer#>

dski is essentially the same function as consol except that input lines are read from a disk file rather than from a terminal. filename names a DEC-10 file. Unlike LISP, file extensions are part of the filename. Thus DETKJO.LOG is represented as DETKJO'.LOG rather than as the dotted pair (DETKJO . LOG).

The location of filename may be specified by adding the project and programmer numbers to the argument list. They should be entered (in octal) as a fraction with the project # as numerator and the programmer # as denominator. If no project/programmer number is specified, the current account is searched for filename. If filename is not found, the DEC-10 Fortran I/O error recovery

routines take over.

Ex: dski:@test'.fil

dski:(test'.fil)

dski:<@test'.fil>

dski:<@test'.dsi 50105/5002>

dski:a where a ==> to same file specification

parse:charstream

The function parse takes a list of characters, such as that produced by consol or dski and returns a list of forms in Daisy's internal representation, the first element of which is a carriage return.

```
Ex: parse:consol:@'% ==>  (
                          %all_("_____"_gars^M
                          %eat):_____cash
                           ^M
                           all ("_" gars eat):cash
                          %^Z
                           ^M
                          )
```

```
parse:consol:@'% ==>  (
                      %once_upon_a_time^M
                       ^M
                       once upon a time
                      %there_was_a^M
                       ^M
                       there was a
                      %^Z
                       ^M
                      )

parse:consol:@'% ==>  (
                      %aaa|^M
                      %bbbb^M
                       ^M
                       aaabbbb
                      %^Z
                       ^M
                      )
```

issue:form

issue takes a form in Daisy's internal representation and returns a list of the characters in that form.

Ex: issue:(_A:B__<1_2_3>) ==> ((A : B _ < 1 2 3 >))

screen:charlist

screen is (almost) the inverse of consol, however, consol compresses blanks that screen cannot restore. It takes a list of characters, writes them at the programmer's console, and returns [] as its value.

NOTE

screen has a known bug that is scheduled to be fixed in the next release - the first element of charlist is lost.

dsko:<charlist>

dsko:<charlist filename>

dsko:<charlist filename project /programmer >

dsko writes the elements of the list of characters charlist to a disk file. If no file is specified, then the default file written to is DSI.DAT. filename and project /programmer are specified as in dski. If the file does not exist, it is created. If it does, dsko appends to it. dsko writes to the specified file deterministically (as a side effect) and returns [] after the write has occurred.

CHAPTER 9

FORM EVALUATION FUNCTIONS

@form

The macro function @ returns form without evaluating it.

```
Ex:  cons:<@A ()>  ==>  (A)
     @remark       ==>  remark
     @x:q          ==>  x:q
     @<a b c>      ==>  <a b c>
```

Evaluation of a list of forms can also be prevented by the use of the parentheses as bracketing characters.

```
Ex:  (a b c)       ==>  (a b c)
     (remark x:q)  ==>  (remark x:q)
```

%form

% is a macro character that specifies that the form following it is to be evaluated.

```
Ex:  let:( (a b) (forty-two a) <@b b %b @a a %a> )
          ==>     (b a forty-two a forty-two !?!)
```

evlst:L

evlst returns a list of the results of evaluating each of the forms in the list/fern L.

Ex: evlst:parse:dski:filnam reads a character stream from the file specified by filnam, parses it into a list of forms, and evaluates each in turn.

NOTE

There are only subtle textual differences between the forms in the next two examples. () and <> have been used carefully to cause the desired evaluation to take place.

```
let:( (x z) (z two)  evlst:< 2:<z x> 2:(z x)
                             2:<x z> 2:(x z)> )
    ==>  (two z !?! two)

let:( (x z) (z two)  evlst:( 2:<z x> 2:(z x)
                             2:<x z> 2:(x z)) )
    ==>  (z x two z)
```

CHAPTER 10

CONDITIONAL FORMS

if:<PRED1 CONSEQ1 PRED2 CONSEQ2 ... PREDN CONSEQN>

if:<PRED1 CONSEQ1 PRED2 CONSEQ2 ... PREDN CONSEQN ALTERN>

if is Daisy's conditional function. Each PREDicate is evaluated in turn (from left to right) until one evaluates to something beside [] or |?|. The value of the CONSEQuent immediately following that PREDicate (or the value of the ALTERNative, if all PREDicates evaluate to [] or |?|) is returned as the value of if. The form

```
if:<A B C D>
```

is Daisy's syntax for the more conventional

```
   if  A  then  B
elseif  C  then  D,
```

while

```
if:<A B C D E>
```

represents

```
if A then B
elseif C then D
else E.
```

Currently, only the formal if:<...> syntax is accepted.

```
Ex: if:<[] @NEVER 1 @DULL [] @MOMENT> ==> DULL
    if:<[] 1 [] 2 [] 3 42>            ==> 42
```

CHAPTER 11

PREDICATES

and:fern

The value of and is [] if any of the fern's "first" elements evaluate to [] or |?|, and true otherwise (if none of them do). Its semantics (in Daisy) are

```
and:L = if:< empty?:L     @true
            empty?:1:L    []
            and:rest:L >.
```

Thus, and looks at successive elements of its argument until either it finds one that is [] (i.e. empty?) or it runs out of argument elements.

and is a function which can be applied to a multiset with very interesting results. For example, consider

```
and:< forever:1  []  [] >
where  forever:X = forever:X.
```

The value of this function call should be [] but won't be since the

evaluation of forever:X proceeds forever, locking out the possibility of finding any of the []s. The invocation

```
and:{ forever:1  []  [] }
```

however, DOES return [] since the evaluation of each element proceeds in parallel, and discovering a [] element isn't precluded by forever:1's computation.

or:fern

The value of or is true if any one of the "first" elements of the fern is not [] or |?|. Otherwise, or returns []. Its semantics in Daisy are

```
or:L = if:< empty?:L  []
            1:L       @true
            or:rest:L >.
```

Like and, or is most powerful when given a multiset argument. Furthermore,

```
or:{ and:{LF1 LF2 LF3} and:{LF4 LF5 LF6} and:{LF7 LF8} }
```

specifies a breadth-first evaluation of the state space

```
<<LF1 LF2 LF3> <LF4 LF5 LF6> <LF7 LF8>>
```

while

```
or:< and:<LF1 LF2 LF3> and:<LF4 LF5 LF6> and:<LF7 LF8> >
```

specifies a depth-first evaluation.

same?:fern

same? checks for pointer equality between the first two elements of the fern.

Ex: same?:<@A first:(A B C)> ==> true
same?:{L L} ==> true
same?:<(1 2 3) (1 2 3)> ==> []
same?:<(a b) a> ==> []
same?:<21 21> ==> true or [] depending on how numbers are implemented in the current release.

atom?:X

The value of atom? is true if X is a number or an identifier and [] otherwise.

Ex: atom?:10 ==> true
atom?:{1 2 3} ==> []
atom?:@A ==> true
atom?:A ==> depends on what A is bound to

empty?:X

empty? returns true if X ==> [] or |?| and returns [] otherwise.

Ex: empty?:[] ==> true

empty?:<> ==> true

empty?:{} ==> true

empty?:A ==> true if A is bound to a null list OR if evaluating A causes an error (e.g. if A is unbound)

empty?:(A) ==> []

empty?:5 ==> []

ltrl?:X ==> true if X is an identifier

==> [] otherwise

nmbr?:X ==> true if X is a number

==> [] otherwise

list?:X ==> true if X is a fern

==> [] otherwise

lt?:<a b> ==> true if a<b, (a and b both numbers)

==> [] otherwise

le?:<a b> ==> true if a<=b, (a and b both numbers)

==> [] otherwise

eq?:<a b> ==> true if a=b, (a and b both numbers)

==> [] otherwise

ne?:<a b> ==> true if a<>b, (a and b both numbers)

==> [] otherwise

ge?:<a b> ==> true if a>=b, (a and b both numbers)

==> [] otherwise

gt?:<a b> ==> true if a>b, (a and b both numbers)

==> [] otherwise

CHAPTER 12

ARITHMETIC FUNCTIONS

neg:N

neg negates N (returns the additive inverse).

```
Ex:  neg:5      ==>   -5
     neg:-5     ==>    5
     neg:-7/10  ==>   7/10
     neg:-3/-2  ==>  -3/2
     neg:4/2    ==>  -4/2
```

inv:N

inv inverts N (returns the multiplicative inverse).

```
Ex:  inv:5      ==>    1/5
     inv:-5     ==>   -1/5
     inv:-7/10  ==>  -10/7
     inv:-3/-2  ==>    2/3
     inv:4/2    ==>    2/4
```

num:N

num returns a number representing the numerator of N.

```
Ex: num:5      ==>  5
    num:-5     ==> -5
    num:-7/10  ==> -7
    num:-3/-2  ==>  3
    num:4/2    ==>  4
```

den:N

den returns a number representing N's denominator.

```
Ex: den:5      ==>  1
    den:-5     ==>  1
    den:-7/10  ==> 10
    den:-3/-2  ==>  2
    den:4/2    ==>  2
```

sgn:N

sgn returns 1, -1, or 0 depending on whether the value of N is greater than, less than, or equal to 0.

```
Ex: sgn:5      ==>  1
    sgn:0      ==>  0
    sgn:-7/10  ==> -1
    sgn:-3/-2  ==>  1
```

quo:N

quo returns a number representing the integral quotient of N's

numerator and its denominator. In the case of N<0, the integral quotient is such that the remainder is always positive.

```
Ex: quo:5      ==>  5
    quo:7/10   ==>  0
    quo:-7/10  ==> -1
    quo:-3/-2  ==>  1
```

rem:N

rem returns a number representing the remainder on division of the numerator of N and its denominator. This remainder is always a positive integer.

```
Ex: rem:5      ==> 0
    rem:7/10   ==> 7
    rem:-7/10  ==> 3
    rem:-3/-2  ==> 1
```

rdc:N

rdc returns a number representing N but reduced to lowest terms.

```
Ex: rdc:5      ==>  5
    rdc:5/10   ==>  1/2
    rdc:-7/10  ==> -7/10
    rdc:-9/-3  ==>  3
```

inc:N

inc returns a number one greater than N.

```
Ex: inc:5       ==>   6
    inc:-5      ==>  -4
    inc:-7/10   ==>  3/10
    inc:7/10    ==> 17/10
    inc:4/2     ==>   3
```

dcr:N

dcr returns a number one less than N.

```
Ex: dcr:5       ==>    4
    dcr:-5      ==>   -6
    dcr:-7/10   ==> -17/10
    dcr:7/10    ==>  -3/10
    dcr:4/2     ==>    1
```

add:<NUM1 NUM2>

add returns the sum, NUM1+NUM2.

```
Ex: add:<1/2 1/2>   ==> 1
    add:<-1/2 1/2>  ==> 0
    add:<3 4>       ==> 7
```

sub:<NUM1 NUM2>

sub returns a number representing NUM1-NUM2.

```
Ex:  sub:<1/2 1/2>   ==>   0
     sub:<-1/2 1/2>  ==>  -1
     sub:<3 4>       ==>  -1
     sub:<3 -4>      ==>   7
```

mpy:<NUM1 NUM2>

mpy returns the product, NUM1*NUM2.

```
Ex:  mpy:<1/2 1/2>   ==>   1/4
     mpy:<-1/2 1/2>  ==>  -1/4
     mpy:<3 4>       ==>    12
     mpy:<3 -4>      ==>   -12
```

div:<NUM1 NUM2>

div returns the quotient, NUM1/NUM2.

```
Ex:  div:<1/2 1/2>   ==>    1
     div:<-1/2 2>    ==>  -1/4
     div:<3 4>       ==>   3/4
     div:<3 -4>      ==>  -3/4
```

sigma:<NUM1 NUM2 ... NUMN>

sigma returns the sum, NUM1+NUM2+...+NUMN.

```
Ex:  sigma:<1 2 3 4 5>  ==>  15
     sigma:{1 1 1 1 1}  ==>   5
     sigma:<1 -1 1 -1>  ==>   0
     sigma:(-1 -1 -1)   ==>  -3
```

pi:<NUM1 NUM2 ... NUMN>

pi returns the product, NUM1*NUM2*...*NUMN.

```
Ex:  pi:<1 2 3 4 5>  ==>  120
     pi:<1 1 1 1 1>  ==>    1
     pi:{1 -1 1 -1}  ==>    1
     pi:(-1 -1 -1)   ==>   -1
```

CHAPTER 13

MISCELLANEOUS FUNCTIONS

$sgt:N/M

$sgt is used to set various system tuning parameters. It takes a rational number N/M as its argument.

N = 0 ==> M = probe count
= 1 ==> M = how strict the printer is
= 2 ==> M = closure sergeant parameter

$trc:N/M

$trc sets the system trace flag. This is the same flag that can be set when Daisy is first entered. $trc is provided mainly for use by APS internal systems programmers, producing a low level trace of the Daisy interpreter.

M is the (octal) unit number of the device to which tracing information should be sent (1 => screen, other unit numbers as per IU DEC-10 installation). N is the logical and of the traces desired (in octal). M/N is entered as a rational number.

```
traces =     1 => stings
             2 => loads
             4 => news
            10 => probes
            20 => probes +
            40 => scheduler
           100 => device handlers
           200 => parser
           400 => printer
          1000 => evlst
          2000 => evlst
          4000 => evlst
         10000 => garbage collector - top level
         20000 => garbage collector - mark phase
         40000 => garbage collector
```

$clk:X

$clk returns X after doing a garbage collection.

Alphabetic Index of System Functions

Bibliography

[1] Friedman,D.P. and Wise,D.S., Applicative Multiprogramming, Technical Report No. 106, Computer Science Department, Indiana University, Bloomington, Indiana (revised: April 1979).

[2] Friedman,D.P. and Wise,D.S., An approach to fair applicative multiprogramming., in Semantics of Concurrent Computation, G. Kahn (ed.), Berlin, Springer (1979), pp.203-226.

[3] Friedman,D.P. and Wise,D.S., Aspects of applicative programming for file systems., Proc. ACM Conf. on Language Design for Reliable Software, ACM SIGPLAN Notices 12:41-85 (March 1977).

[4] Friedman,D.P. and Wise,D.S., Aspects of applicative programming for parallel processing., IEEE Transactions on Computers C-27(4):289-296 (April 1978).

[5] Friedman,D.P. and Wise,D.S., CONS should not evaluate its arguments., in Automata, Languages and Programming, S.Michaelson and R.Milner (eds.), Edinburgh, Edinburgh University Press (1976), pp. 257-284.

[6] Friedman,D.P. and Wise,D.S., Functional combination., Computer Languages 3(1):31-35 (1978).

[7] Friedman,D.P. and Wise,D.S., An indeterminate constructor for applicative programming., Seventh Annual Symposium on Principles of Programming Languages, (January 1980), pp.243-250.

[8] Friedman,D.P. and Wise,D.S., A note on conditional expressions., Comm. ACM 21(1):931-933 (November 1978).

[9] Friedman,D.P. and Wise,D.S., Output driven interpretation of recursive programs, or writing creates and destroys data structures., Information Processing Letters 5(6):155-160 (December 1976)., Erratum: Information Processing Letters 9(2):101 (August 1979).

[10] Friedman,D.P. and Wise,D.S., Unbounded computational structures., Software - Practice and Experience 8:407-416 (1978).

[12] Grismer,T.M., Solving common programming problems with an applicative programming language., M.S. thesis, Indiana University, Bloomington Indiana (1980).

[13] Johnson,S.D., Circuits and systems: implementing communication with streams., Technical Report No. 116, Computer Science Department, Indiana University, Bloomington, Indiana (October 1981).

[14] Johnson,S.D., An interpretive model for a language based on suspended construction., M.S. thesis, Indiana University, Bloomington (1977).

[15] Smoliar,S.W., Using applicative techniques to design distributed systems., Proc. IEEE Symposium on Specifications of Reliable Software., 1979, pp.150-160.

[16] Friedman,D.P. and Wise,D.S., Fancy ferns require little care., Technical Report No. 106, Computer Science Department, Indiana University (March 1981).