Circuits and Systems: Implementing Communication with Streams*

by

Steven D. Johnson

Computer Science Department

Indiana University

Bloomington, Indiana 47405

TECHNICAL REPORT NO. 116

CIRCUITS AND SYSTEMS:

IMPLEMENTING COMMUNICATION WITH STREAMS

Steven D. Johnson

Revised July 1982

*Research reported herein was supported in part by the National Science Foundation under grant number MCS 77-22325.

To appear in the Proceedings of the 10th IMACS World Congress on Systems Simulation and Scientific Computation to be held in Montreal, Canada, 8-13 August: Edited by W.F. Ames and R. Vichnevetsky (Volume 5) Parallel and Large Scale Computers: Systems, Applications and Performance Evaluation.

Circuits and Systems: Implementing Communication with Streams[1]

Steven D. Johnson

Indiana University Computer Science Deptartment
101 Lindley Hall
Bloomington, Indiana, 47405

Models for a basic electronic circuit and a small time sharing system are implemented applicatively in *Daisy*, an output-driven list processing language. Rather than using shared variables to handle communication, these recursively defined systems use data streams, interpreted as changing over time.

1. Introduction

Friedman and Wise propose applicative language constructs[8] and a new data constructor[11] to address concurrency and indeterminism. The programming language *Daisy*, being developed at Indiana University, incorporates these features. Its novelty lies in the fact that it is purely *output-driven*. All data creation, hence all computation, is on a call-by-need basis. Other terms for this approach to computation are *delay rule*,[24] *demand driven computation*,[17] *lazy evaluation*,[1] *lenient functions*,[25] and *suspending evaluation*.[7] It is assumed that the reader is familiar with this idea, but not necessarily its practice; that is the subject of this report.

Algorithms are implemented below that are distributed in nature. Since communication in such "circuits" tends to be regarded as mutual access to a shared variable (the wires connecting components[19]), programming them applicatively (without side effects) is a challenge. We shall instead implement the communication links as *streams* (non-finite lists), viewed here as functions-over-time. This is not an original approach; other proponents of applicative style adopt the same technique.[4, 13, 16, 2]

The experience of implementing these circuits refines an understanding of technique and semantics, for which we pay very little syntactically. We use the only tool at hand – systems of recursion equations – and find that it is nearly ideal for describing some circuits. Daisy's interpreter "solves" recursive systems lazily. Consequently, we are able to build and compute with non-finite as well as finite objects, without having to make a distinction between them. Here, we are concerned with stream-like objects – non-finite sequences of finite elements[2].

Section 2 discusses Daisy's syntactic eccentricities. (Appendix A is a review of the language; Appendix B lists those primitive functions used in this report.) Sections 3, 4 and 5 each present a program and its implementation. Since we are concerned with a variety of issues about these programs, each section concludes with a number of loosely connected discussions. Section 6 is a summary.

This report has two recurrent themes. First is the idea of program-as-*schematic*. Circuit drawings are a venerable algorithmic language, having evolved much longer than other programming languages. Expositions of ordinary algorithms usually start with some kind of picture – a state machine, flow diagram, *etc* – but these descriptions are pure syntax. It is becoming increasingly clear that the meanings of these pictures can be found in systems of equations, just as finite-state processes are described by their transition matrices. Sections 4 and 5 model examples from two disparate areas of computer science: circuits and operating systems. Yet their schematics reveal their structural similarity.

Second, the duality of process and data, reflected in the manipulation of streams, leads to a "type conflict". At lower levels, the content of a stream is viewed as the output of a function-over-time. Some functions describe *components*; a value change on their input stream results, a short time later, in a value change on their output stream. More complex functions are combinations of components. They simply express connectivity (that is, they describe schematics) and manipulate streams as objects.

2. Elements of Daisy's Syntax

For more details, see Appendix A.

Functions are denotable objects in Daisy. The expression \x.E denotes the function "λx.E", whose value when applied to an argument *a*, is E's value with all occurrences of x replaced by *a*.

All functions take one argument, which may be "structured" by a *pure data* description. This is analogous to a Pascal RECORD declaration or a Fortran EQUIVALENCE statement. Since the interpreter is output-driven, arguments are not *compelled* to match their declared structure, however, unless and until they are accessed according to it. Users partial to LISP might define[3]:

```
car:[a ! d] ⇐ a.
cdr:[a ! d] ⇐ d.
cons:[a d] ⇐ <a ! d>.

or:L ⇐ let [La ! Ld] = L in
         if null?:L then <>
         else if La then La
         else or:Ld.
```

Sequences of the form "<...>" are *determinate* in the sense that they specify an order for their elements. *Multisets*, which are sequences surrounded by braces: "{...}", specify their elements, but not their order. When an order is needed, one is selected based on the relative cost of computing the elements. Divergent elements never precede convergent ones.[6, 10, 11]

An exclamation point is the list concatenator. An asterisk is a primitive stream builder: <x *> is equivalent to

rec let L = <x ! L> **in** L.

Both expressions yield a non-terminating sequence whose elements are identical.

Examples:

<x *> **where** x = 5 *evaluates to* [5 5 5 . . .]
<1 2 3 *> *evaluates to* [1 2 3 3 3 . . .]

Structures are applied to arguments in a manner analogous to vector arithmetic [10,14]. The argument, assumed to be in "row major" format (a list of rows), is *transposed.* The function-vector is applied coordinate-wise to the transposed argument. The key word "#" deforms argument-matrices; #'s are skipped during transposition.

Examples:

dotproduct:[V1 V2] ⇐ sum:[multiply*]:<V1 V2>.

matrixadd:[M1 M2] ⇐ [[add *] *]:<M1 M2>.

```
[cons cons]:[
[ a   # ]
[ #   b ]      is equivalent to <cons:[a c] cons:[b d]>.
[ c   d ] ]
```

wire:S ⇐ <(\x.x)*>:S.

The reader should pause here to become comfortable with the definition of wire, which is used throughout. We would like to identify functions that return streams of sequences with functions that return sequences of streams. (This is the "type conflict" alluded to in the Introduction.) Functional combination transposes the (possibly non-finite) argument for us; all that is needed is to apply the identity function \x.x repeatedly. If X contains no #'s, wire:wire:X is equivalent to X.

3. Generating Streams

We may think of streams either as objects or processes. Correspondingly, there are at least two ways to build them: with "functional recursion" and with "data recursion"[9] This is primarily a syntactic distinction, although an implementation may favor one technique over the other. To illustrate, consider a stream of increasing integers. The non-finite sequence can be expressed as the result of a function:

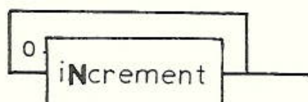
rec N:0 **where** N:i ⇐ <i ! N:add:<i 1>>.

However, the expression

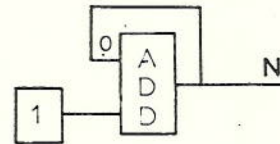
rec N **where** N = <0 ! [add*]:<N <1*>>>.

yields the same result.

Regardless of their semantic equivalence, there is a subjective difference between these expressions. In the first expression N is a stream *transformer,* a function that iteratively computes its output.



In the second expression, N is just the name of a stream, specifying the connectivity of an expression-circuit.



The second schematic is composed of more primitive components:

1. A *constant* stream that supplies a source of 1's to...
2. ...the component ADD, a stream transformer that distributes the primitive add operation pair-wise on its input streams, [add*]:<U V>.

Discussion: Implementation[4] (Figure 1)

Figure 1 shows an interactive session with Daisy in which both stream builders are implemented. The operator requests an infinite stream of integers and one is printed: the program must be aborted. (The host is a Digital Equipment computer, where the system interrupt character is EXT, typed "^C".) Daisy is restarted, and the alternative expression yields the same behavior.

Discussion: Call-by-need

Still in Figure 1, Daisy is entered a third time. This time we ask for an element of the integer stream. Since all that is needed (for output) is a single element; interaction can continue. Normal termination follows, brought about by typing an end of file character on the keyboard stream. It is not the fact that an object is infinite that ties up the system, rather that the system has been asked to *print* an infinite object.[12]

Discussion: Mapping

The form of functional combination implemented in Daisy, a LISP-like "mapping" operation,[8] subsumes some commonly used recursion patterns. For example, it contains an implicit conditional to check for list termination. Mapping has always been the preferred way to deal with lists and arrays, and it is the preferred way to deal with streams, too. However, this form of combination is not the only way to map functions.[3, 15, 22] Often a stream transformer must carry some history (state) across iterations. When this is the case, we shall see occurrences of functional circularity (see the functions SELECT and MERGE in Section 5).

4. Feedback

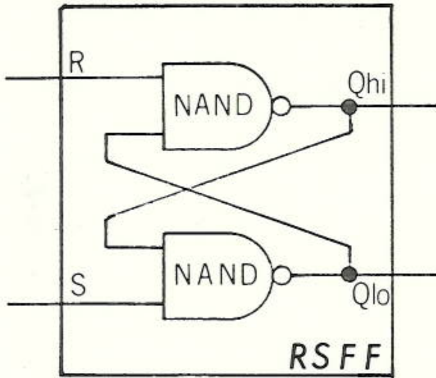
In the previous section a system of one recursive function (data) symbol is associated with a schematic of one component. Recursion is expressed as a feedback loop in the circuit. This generalizes immediately to systems of more than one recursive symbol. In this section we program a model of a "reset/set flip-flop".

```
% Daisy
& rec:( N \(i . <i ! N:add:<i 1>> )
&        N:0   )
( (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 2
6 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 5
0 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 7
4 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89^CInterrupt
%
%  Daisy
& rec:( N <0 ! <add*>:< <1*> N>>
&        N  )
( (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 2
6 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 5
0 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 7
^CInterrupt
%
%  Daisy
&  700:rec:( N <0 ! <add*>:< <1*> N>>
&            N  )
( 699
&

& ^D
& ^D

)
DSI exit.
```

Figure 1
Interaction with Daisy



The components are identical:

NAND:[L R] ⇐ [nand*]:<L R> **where**

nand:[u v] ⇐ **if** eq?:<u 0> **then** 1
else if eq?:<v 0> **then** 1 **else** 0.

The "logic" function nand takes truth levels, 1 or 0, and interprets 1 as *true*. NAND , like ADD in Section 3, distributes nand across a pair of streams.

We would like the *program* RSFF to be a transcription of its schematic:

idealRSFF:[R S] ⇐ **rec** <Qhi Qlo>
where
Qhi = NAND:<R Qlo>
Qlo = NAND:<S Qhi>.

But this expression does not work. The mutually dependent outputs, Qhi and Qlo, are *deadlocked.* As is demonstrated in Figure 2, deadlock is alleviated by initializing both streams.

RSFF:[R S] ⇐ **rec** <Qhi Qlo>
where
Qhi = <1 ! NAND:<R Qlo>>
Qlo = <0 ! NAND:<S Qhi>>.

Discussion: Implementation of RSFF (Figure 2)

Figure 2 shows an interactive test of RSFF in Daisy. Wire (defined in Section 2) converts RSFF's two outputs to a single stream of pairs so they can be printed in parallel. To test the function, two channels are established with the operator's keyboard. To the prompts "R" and "S", streams of 1's and 0's are typed. Each output line reflects the two keyboard inputs immediately preceding it.

Line (1): RSFF retains its state (the values on Qhi and Qlo) as long as its inputs (the values on R and S are held high.

Line (2): A pulse on S (a burst of 0's) leads to a change in state. After a moment of confusion, Qhi and Qlo reverse their values.

Line (3): A spike on one of the input streams (a single 0) is captured by the system. The outputs start oscillating...

Line (4): ... and continue to do so until a "reset" is asserted.

Line (5): RSFF issues ambiguous output (Qhi = Qlo) if both R and S are asserted...

Line (6): ... and becomes "metastable" if R and S return to high values simultaneously.

```
& format: RSFF:< keyboard:@R  keyboard:@S >

R 1 1 1 1 1 1 1 1 1
S 1 1 1 1 1 1 1 1 1
 ((1 0) (1 0) (1 0) (1 0) (1 0) (1 0) (1 0) (1 0) (1 0)
R 1 1 1 1 1 1 1 1 1
S 1 1 1 0 0 0 1 1 1
 (1 0) (1 0) (1 0) (1 0) (1 1) (0 1) (0 1) (0 1) (0 1)
R 1 1 1 1 0 1 1 1 1
S 1 1 1 1 1 1 1 1 1
 (0 1) (0 1) (0 1) (0 1) (0 1) (1 1) (0 0) (1 1) (0 0)
R 1 1 1 0 0 0 1 1 1
S 1 1 1 1 1 1 1 1 1
 (1 1) (0 0) (1 1) (0 0) (1 1) (1 0) (1 0) (1 0) (1 0)
R 1 1 1 0 0 0 0 0 0
S 1 1 1 1 1 1 0 0 0
 (1 0) (1 0) (1 0) (1 0) (1 0) (1 0) (1 0) (1 1) (1 1)
R 0 0 0 1 1 1 1 1 1
S 0 0 0 1 1 1 1 1 1
 (1 1) (1 1) (1 1) (1 1) (0 0) (1 1) (0 0) (1 1) (0 0)
R
```

Figure 2
An Interactive Test of RSFF

Discussion: Idealized Behavior

Both of RSFF's outputs vary over time, subject to the condition of its inputs. The graininess of the model depends on how time is quantized in its streams. RSFF's behavior is similar to that of a real flip-flop. It is ambiguous on invalid inputs, and in extreme conditions goes into a kind of metastable state.[20] Because it is a discrete model, there are no thermal fluctuations or manufacturing disparities to attenuate its oscillation. Nor does it manipulate voltage levels in detail. Some of these discrepancies can, of course, be eliminated by refined modelling.[5, 21]

Discussion: The Principle of Stream Cons-*ervation*

Disregarding any notion of causality, RSFF works best when its outputs are consumed at the same "rate" as its inputs are supplied. Were RSFF to be used as a component in a system that, over the long run, needed two Qhi values for every Qlo value, we would have to adjust RSFF's code accordingly; and we could do so for any known ratio of rates. This has the flavor of an equilibrium assumption, and is an analytic property of RSFF's computation. A sketch of a proof that RSFF produces no faster than it consumes uses the circuit's structure.

RATE(Qhi)
= MIN(RATE(R), RATE(Qlo))
= MIN(RATE(R), MIN(RATE(S), RATE(Qhi))
= MIN(MIN(RATE(R), RATE(S)) RATE(Qhi))
$\leq$ MIN(RATE(R), RATE(S))

The first equality comes from axiomatic properties of primitive components. The second is by symmetry of the circuit. The rest come from facts about MIN.

Discussion: Coercion

This system contains both stream transformers and stream handlers. The functions RSFF and NAND are just connectivity expressions; they steer their stream-arguments to more primitive components. The expression [nand *]:X uses the underlying system (*via* functional combination) to compose two stream-arguments into one argument-stream-of-pairs, which in turn is transformed by sequential applications of nand.

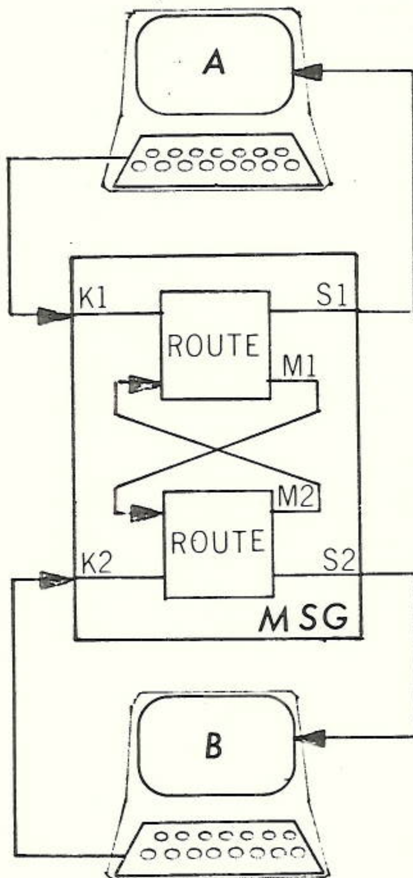
Discussion: Stream Initialization

The need to put initial values on streams is analogous to the need to code termination conditions in numerical functions. Since our real interest is in RSFF's running behavior, it seems like a bothersome detail. Of course these values form the inductive basis for any assertion about the program, so they are necessary. For example, to prove the invariant: "Qhi differs logically from Qlo as long as R and S remain high," Qhi and Qlo must be different to begin with.

5. Indeterminacy

RSFF is "synchronized" by stream construction. While this property can be preserved *within* a system, there is no guarantee that its inputs and outputs behave so well. This section describes a technique to handle indeterminacy. Suppose we are to write a system with the hypothetical specification:

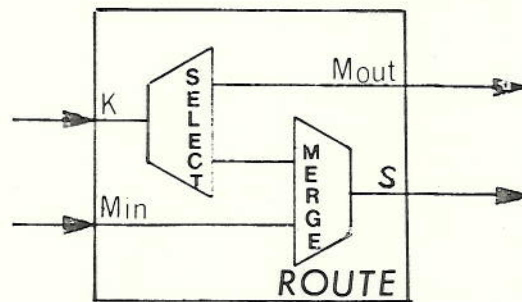
"...MSG is a full duplex message sending system for two computer terminals, A and B. Input from A's (respectively B's) keyboard, K_1 (K_2) is echoed on A's (B's) screen, S_1 (S_2). However, when K_1 issues a "send", the ensuing form should be displayed *in a timely fashion* on S_2 (and conversely for K_2 and S_1)...."



The schematic for MSG has roughly the same connectivity as RSFF's, except that its ROUTE components have two outputs instead of one – for screen display and for message feedback.

```
MSG:[KA KB] ⇐ rec <SA SB>
  where
    [SA MA] = ROUTE:<KA MB>
    [SB MB] = ROUTE:<KB MA>.
```

MGS's components are themselves circuits. The function ROUTE takes keyboard (K) and message input (Min), combines them, and delivers screen output (S) and message feedback (Mout).




```
ROUTE:[K Min] ⇐ rec <S Mout>
  where
    [Mout Dplx] = wire:<<@On #> ! SELECT:K>
    S = MERGE:<Dplx Min>.
```

ROUTE takes care of stream initialization, as well as the extraction of distinct streams from SELECT. MERGE manages indeterminacy in a manner similar to that of[11] .

```
SELECT:K ⇐
  rec let [Kh ! Kt] = K
          [Kth ! Ktt] = Kt
  in
    if same?:<Kh @send>
      then <<# ! <@? Kth>> ! SELECT:Ktt>
      else <<Kh ! #> ! SELECT:Kt>.

MERGE: [L R] ⇐
  let [La!Ld] = L
      [Ra!Rd] = R
  in
     (\[x].x):
       { strictify:<La <La ! MERGE:<R Ld>>>
         strictify:<Ra <Ra ! MERGE:<L Rd>>> }.
```

Of concern is whether MSG meets the informal specification "...in a timely fashion...". Not only must MSG avoid deadlock (a data dependency like that in idealRSFF, Section 4), it must avoid *lockout* as well. Lockout would result if the system were to wait deterministically for input from a specific terminal.

Discussion: Implementation of MSG (Figure 3)

Lines beginning with the prompt "A" (respectively "B") are the keyboard input for terminal A (B). The remaining lines show the output to terminal A's screen. B's screen is not shown.

We may think of the keyboard inputs as concurrent. B only sends messages, which appear on A's screen prefixed with the character "?". A's first entries ("hm hm hm") are echoed on his screen with B's first messages interspersed. As A joins in the round, B's next messages are coming through. This continues as long as B transmits to A's screen.

```
&amp; \((SA SB).SA): MSG:< keyboard:@B  keyboard:@A >

B send ROW send ROW send (ROW YOUR BOAT)
 (On
A hm hm hm hm
 hm (? ROW) hm (? ROW) hm hm (? (ROW YOUR BOAT))
A row row row your boat
B send GENTLY send (DOWN THE STREAM)
B send MERRILY send MERRILY send MERRILY send MERRILY
 row (? GENTLY) row row (? (DOWN THE STREAM)) your boat (? MERRILY)
A gently down the stream
 gently (? MERRILY) down (? MERRILY) the stream (? MERRILY)
A merrily merrily merrily merrily
B send LIFE send IS send BUT send (A DREAM)
 merrily merrily (? LIFE) merrily (? IS) merrily
A life is but a dream
B HM HM HM HM
 (? BUT) life (? (A DREAM)) is but a dream
A
```

Figure 3
An Interactive Test of MSG

The Role of "#"

The system avoids deadlock because it is internally synchronous: the multivalued functions always produce values for each of their outputs. It is at the innermost component, MERGE, where the indeterminacy is resolved. SELECT's stream of pairs is coerced by ROUTE to a pair of streams. But SELECT creates pairs containing #'s, which functional combination skips during transposition. So while SELECT is synchronously producing its stream, that stream is deformed to present "real time" data to MERGE.

Discussion: Is MSG Fair?

A heuristic discussion on the fairness of MSG has the form of a subgoal induction.[23] At any level, we shall assert that if the system is unfair, it is the fault of its components.

The *function* MSG is fair by symmetry. Its connectivity precludes favoritism for either keyboard.

ROUTE is a function composition. Its components are connected, more or less in series, so if either induces lockout, ROUTE may. SELECT waits for its input to converge, but SELECT *has* only one input! Surely it is all right for a function to wait for its only source of information.

This brings us to MERGE. The form:

(\[x].x):{ strictify:<u v> strictify:<x y> }

is analogous to the *guarded command*

$\underline{\text{if}}\ u \rightarrow v\ [\,]\ x \rightarrow y\ \underline{\text{fi}}$

Identifiers $\underline{x}$ and $\underline{y}$ play the role of *input guards*.[14] MERGE is fair modulo the "{ ... }" construction.[10] In fact it is probably fairer. MERGE swaps its inputs occasionally (note carefully the recursive calls). Suppose that the underlying system is neither fair nor malicious, rather that it is *biased* – favoring multiset arguments according to their textual order. Then by switching that order from time to time MERGE ameliorates the bias.

6. Summary

In Sections 4 and 5 models are implemented for a fundamental electrical circuit and a small timesharing system. That they were implemented in the same language speaks not so much for the language itself as for the similarity of their schematic structure. Both are circuits of two autonomous components; both have about the same feedback relationship.

The translation from schematic to program is testimony to the expressive power and usefulness of recursive systems of equations. In these systems, communication is achieved by encoding time in streams. Since the interpreter used is output-driven, streams are built without special language primitives.

For completeness, designers of applicative languages must choose an interpretation for the application of a structure to an argument. The form of combination used in Daisy works well in these examples because it provides the right transformation to implement the connectivity of schematic programs.

Where the behavior of input producers and output consumers is known, systems are synchronized by the rate at which they build streams. Multisets can be used to model lockout avoidance in systems with indeterminate input behavior.

Notes

[1]Research reported herein was supported (in part) by the National Science Foundation under grant number MCS77-2222325.

[2]Daisy's infinite lists are not "legitimate" streams (in the sense of[18]) since they are not strict in their elements. But until Section 5 there is little need to make a distinction.

[3]The versions of car and cdr defined here, however, are not strict, since cons is suspended. Consequently, the laws:

$$\mathrm{car{:}cons{:}}\langle a\ b\rangle \equiv a$$
$$\mathrm{cdr{:}cons{:}}\langle a\ b\rangle \equiv b$$

hold, even if *a* or *b* diverges.

[4]Daisy's full parser is still under development. The syntax of programs run to generate the Figures differs from what is presented in this report.

References

1. P. Henderson and J.H. Morris, Jr., "A lazy evaluator," *Third ACM Symposium on Principles of Programming Languages*, pp. 95-103 (1976).
2. Ed Ashcroft and Bill Wadge, "Lucid, a nonprocedural language with iteration," *Comm. ACM* **20**(7) pp. 519-526 (July, 1977).
3. J. Backus, "Can programming be liberated from the von Neumann style?," *Comm. ACM* **21**(4) pp. 613-641 (August 1978).
4. W. H. Burge, *Recursive Programming Techniques*, Addison-Wesley, Reading, Pa. (1975).
5. L. Cardelli, "Analog processes," *Proceedings of the Ninth Symposium on Mathematical Foundations of Computer Science*, pp. 181-193 Springer-Verlag, (1980).
6. D. P. Friedman and D. S. Wise, "Applicative multiprogramming," Technical Report No. 72, Indiana Univ. Computer Science Dept., Bloomington, Indiana (revised: April 1979).
7. D. P. Friedman and D. S. Wise, "CONS should not evaluate its arguments," pp. 257-284 in *Automata, Languages and Programming*, ed. S. Michaelson and R. Milner, Edinburgh University Press, Edinburgh (1976).
8. D. P. Friedman and D. S. Wise, "Functional combination," *Computer Languages* **3**(1) pp. 31-35 (1978).
9. D. P. Friedman and D. S. Wise, "Unbounded computational structures," *Software - Practice and Experience* **8** pp. 407-416 (1976).
10. D. P. Friedman and D. S. Wise, "An approach to fair applicative multiprogramming," pp. 203-226 in *Semantics of Concurrent Computation*, ed. G. Kahn, Springer-Verlag, New York (1979).
11. D. P. Friedman and D. S. Wise, "An indeterminate constructor for applicative programming," pp. 243-250 in *Seventh Annual Symposium on Principles of Programing Languages*, (January 1980).
12. D. P. Friedman and D. S. Wise, "A note on conditional expressions," *Comm. ACM* **21**(1) pp. 931-933 (November 1978).
13. Peter Henderson, "Purely functional operating systems," pp. 177-192 in *Functional Programming and its Applications*, ed. J. Darlington, P. Henderson, and D.A. Turner, Cambridge University Press, Cambridge (1982).
14. C. A. R. Hoare, "Communicating sequential processes," *Comm. ACM* **21**(1) pp. 666-677 (August 1978).
15. K. E. Iverson, "Notation as a tool of thought," *Comm. ACM* **23**(8) pp. 444-469 (August, 1980).
16. G. Kahn and D. MacQueen, "Coroutines and networks of parallel processes," *IFIP 77*, pp. 933-938 North-Holland, (1977).
17. R. M. Keller, T. Lindstrum, and S. Patil, "A loosely-coupled applicative multi-processing system," *Proc. National Computer Conference, 1979*, pp. 613-622 (1979).
18. P. J. Landin, "A correspondence between ALGOL 60 and Church's lambda notation - part I," *Comm. ACM* **8**(2) pp. 89-101 (February, 1965).
19. N. A. Lynch and M. H. Fischer, "On describing the behavior and implementation of distributed systems," *Theoretical Computer Science* **13**(1) pp. 17-43 (January, 1981).
20. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts (1980).
21. R. Milner, "On relating Synchrony and Asynchrony," Technical Report No. CSR-75-80, Univ. of Edinburgh, Edinburgh (1980).
22. James H. Morris, "Real programming in functional languages," pp. 177-192 in *Functional Programming and its Applications*, ed. J. Darlington, P. Henderson, and D.A. Turner, Cambridge University Press, Cambridge (1982).
23. J. H. Morris and B. Wegbreit, "Subgoal induction," *Comm. ACM* **20** pp. 209-222 4, (April, 1977).
24. J. Vuillemin, "Correct and optimal implementations of recursion in a simple programming language," *Journ. Computer and System Sciences* **9**(3) pp. 332-354 (1974).
25. C. Wadsworth, *Semantics and Pragmatics of Lambda-calculus*, Ph.D. dissertation, Oxford (1971).

Appendix A
Daisy Syntax

EXP ::= *ATOM* ‖ *FERN* ‖ *APPL* ‖ *FNTN* ‖ *COND* ‖ *SYS*
EXP ::= (*EXP*) ‖ @ *EXP*

The six forms of expression are described below. Parentheses are used for parser direction. The special symbol @ means "quote", that is, inhibit evaluation.

ATOM ::= *LTRL* ‖ *NMBR*

Numerals are expressed and represented as rational numbers. They evaluate to themselves. ***Literals*** are strings of characters and evaluate to their bindings in the current environment.

FERN ::= [*ELST*] ‖ < *ELST* > ‖ { *ELST* } ‖
ELST ::= *empty* ‖ *EXP* * ‖ *EXP* ! *EXP* ‖ *EXP ELST*

A ***fern*** is a list specification. The three enclosure symbols express progressively weaker specifications of content and order. The [...] construct is a structural quotation; it evaluates to precisely the list expressed. The <...> construct evaluates to a list of ***values*** in the order specified. However, computation of these values does not take place unless and until the list is accessed.

Ferns of the form {...} specify content but not order. The interpreter chooses an order at run-time, by evaluating the elements concurrently. The ordering of the element values list depends how fast they converge.

The exclamation point is the list concatenation operator; an asterisk denotes a stream of identical elements.

APPL ::= *EXP* : *EXP*

The left-hand expression in an ***application*** is interpreted as a function and applied to the argument on the right. If the function is primitive, it is executed by the interpreter. For example, numeric functions denote list access, so 5:x evaluates to x's fifth element. Some of the literals reserved to name primitive functions are shown in Appendix B.

If the function-expression evaluates to a list, the interpreter first transposes the argument, which is assumed to consist of a list of "rows". Any instances of the keyword "#" are removed during transposition. The elements of the function are then applied coordinate-wise to the transposed argument's "columns".

FNTN ::= \ *EXP* . *EXP*

A ***function*** is analogous to a lambda-expression. When applied, the function's ***formal argument*** (to the left of the dot) is superimposed on the actual argument of the application. This associates identifiers in the formal argument to their relative positions in the actual argument. The association is ***suspended***, and so is not enforced until the binding for the identifier is actually used in the computation.

The value of a function is its ***closure***, a non-printable object that saves the environment in effect when the function was created. All free variables are bound in the closure's environment, making the language lexically scoped.

COND ::= **if** *EXP* **then** *EXP* **else** *EXP*

The ***predicate*** (if part) is evaluated. If the result is non-null the conditional returns the value of its ***consequent*** (then part). Otherwise, the value of the ***alternate*** (else part) is returned. Evaluation of the consequent and alternate are deferred until the predicate converges.

SYS ::= *BODY* ‖ **rec** *BODY*
BODY ::= **let** *DCL* **in** *EXP* ‖ *EXP* **where** *DCL* .
DCL ::= *empty* ‖ *DFN DCL*
DFN ::= *EXP* = *EXP* ‖ *LTRL* : *EXP* <= *EXP*

A typical Daisy program is a system of defining equations, followed by an expression to be evaluated in that system. If the defining equations are not recursive, they are equivalent to a lambda-expressions. That is,

let (x = a) (y = b) (z = c) in e

is equivalent to

(\ [x y z] . e):<a b c>

The "let...in" and "where..." forms have equivalent meanings.

Systems preceded with the keyword **rec** are recursively defined.

The defining equations may define either functions or data; their left hand sides may be either atomic or lists. The symbol ⇐ makes function definitions easier to read. One may write

F:x ⇐ e.

rather than

F = \ x . e.

Global assignments are allowed at top level. Thus the operator can extend the set of basic operators with definitions of his own.

Appendix B
Operators

The following primitive Daisy operators are used throughout this report.

add — Returns the arithmetic sum of the first two elements of its argument.

console — Takes a single character argument and establishes an independent channel to the operator's keyboard, using that character as a prompt. Returns a stream of characters.

eq? — A test for numeric equality.

format — A user defined function that inserts carraige returns in a stream. Used in Figure 2.

keyboard — A user defined function to convert operator input to a useful format. In Section 4,

```
keyboard:p ⇐ filter:parse:console:p where
  filter:S ⇐ let [Sh ! St] = S in
    if null?:S then <> else
    if nmbr?:Sh then <Sh ! filter:St>
    else filter:St.
```

In Section 5,

```
keyboard:p ⇐ parse:console:p.
```

nmbr? — A test for a numeric argument.

null? — A test for the empty list.

parse — Daisy's parser from character stream to internal representation of expressions.

same? — A test for reference equality.

strictify — Functions cannot test whether a value is suspended, but can exploit Daisy's interpreter to force a value into existence.

```
strictify:<x y> ⇐ if x then y else y.
```