Digital Design in a Functional Calculus

By

Steven D. Johnson
Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 185

Digital Design in a Functional Calculus
by
Steven D. Johnson

November, 1985

This material is based on work supported by the National Science Foundation under grant number DCR 84-05241.

Digital Design in a Functional Calculus

Steven D. Johnson

Indiana University
Computer Science Department
Bloomington, Indiana
U.S.A.

A technique is presented for deriving synchronous-system descriptions from recursive function definitions, using correctness-preserving transformations. The use of functional abstraction to address control and communication is illustrated in a small example. The relationship of this technique to conventional digital design methodology is discussed.

Introduction

This paper develops the thesis that functional notation is natural for the description of digital circuits and that functional calculus is an appropriate basis for digital engineering. Standard function transformation techniques are adapted to the problem of digital circuit synthesis. Functional programmers will find familiar transformation methods applied in an unfamiliar setting; digital engineers will recognize a structured design technique cast in a functional notation. This material is intended for the union of these two groups, not the intersection. The presentation is informal but involves disparate formalisms.

The term "synthesis" is used above according to Manna's definition [4]: *...the theory for constructing ...[implementations] that are guaranteed to be correct [with respect to their specifications] and therefore do not require debugging or verification.* The emphasis here is on "theory", not "constructing". Before automation is attempted, it should be shown that an engineering methodology is palatable and pervasive. It must reflect intuitions about the implementation realm because synthesis requires intelligent guidance. The synthesis process should yield correct implementations, not just correct descriptions of implementations—a distinction that has more bearing in hardware than in software. The technique presented here borrows from an established methodology for software engineering and is also related to practical digital design techniques. However, the scope of its applicability in practice has not been demonstrated.

Manna uses "verification" in reference to logical correctness. In hardware, this term encompasses analytic questions concerning the satisfaction of physical design constraints, testability, presence of spurious electrical properties, and so forth. These kinds of problems are not considered in this paper.

The synthesis theory used here is based on transformation, what Darlington and Burstall describe as "an inference system in which the sentences are recursion equations" [2]. A problem is stated in the form of a recursive function definition, or *specification*. The goal is to systematically derive a circuit description using correctness-preserving transformations. Whether the specification itself is right or wrong is immaterial; the object is to obtain a correct circuit relative to the problem statement.

The transformation target is a recursion scheme (or pattern) that characterizes synchronous digital systems. Syntactically, this target is a schematic in linear form; it states the components and the connectivity of the corresponding circuit. The characteristic scheme is restrictive, but it can be mechanically derived from

To appear in: Workshop on Formal Aspects of VLSI Design, Edinburgh, U.K., 30 June – 2 July, 1985, G. J. Milner and P. A. Subrahmanyam (eds.), (North-Holland, Amsterdam).

the larger class of *iterative* specifications. In essence, iterative functions are the language of structured digital design.

Transformation of arbitrary specifications to iterative form (also known as "recursion removal") cannot be automated in the most general sense [12]. This problem is central to the use of functional calculus to describe computation and has been studied extensively. The explicit treatment of control through the use of *continuations* is a paradigm for the methodology. An aspect of the implementation is introduced as a functional abstraction. The abstraction is then replaced by an appropriate representation, whose interpretation is added to the specification. This process continues until specification is sufficiently concrete.

One advantage of using functional notation in design is its executability. The engineer can experiment directly in the design language, making the precarious translation to simulation packages and other modeling vehicles, in principle, unnecessary. In software the analogy is to use a modeling vehicle, such as Lisp, to quickly develop, explore, and refine a prototype—or "executable specification"—from which an efficient program derives. This work uses a "lazy" Lisp-like language called Daisy, which permits both functions and data to be defined recursively. The normal-order semantics of a lazy language is preferred because of its adherence to the mathematical meaning of functional notation. *Data recursion* is especially useful for modeling circuits; non-terminating digital behavior is readily represented by non-finite lists, sometimes called *streams*. Each of the functional expressions shown in this paper is a stylized Daisy program, but none is peculiar to it. Laziness and streams are easily added to any language that properly provides for functional values [1, 3].

After a review of notation, a simple model of digital behavior is presented. The main sections illustrate the synthesis technique with an example. The conclusion discusses the relationship to conventional design methods and illustrates the use of functional languages in experimentation.

Functional Specifications

Specifications are built from constants, identifiers, operations, applicative terms (e.g. $f(e_1, \ldots, e_n)$), conditional expressions (e.g. $p \rightarrow e_0, e_1$), and function definitions (e.g. $F(x_1, \ldots, x_n) \Leftarrow e$). A term is *simple* if it contains no defined function symbols. The notation is embellished with syntax for making specifications easier to read. One might specify the *factorial* function as

$$Fac(n) \Leftarrow zero?(n) \rightarrow 1,\ multiply(n,\ Fac(decrement(x)))$$

A functionally equivalent version of *Fac* is given by

$$\begin{aligned} &F(n, 1) \\ &\quad where \\ &\quad F(x, y) \Leftarrow zero?(x) \rightarrow y,\ F(decrement(x), multiply(x, y)). \end{aligned}$$

This version is *iterative*: each instance—the single instance in this case—of a defined-function symbol is tail-recursive (outermost) [6]. Specifications can have an operational interpretation beyond the functions they define. The iterative version of *Fac* is a better *program* because it executes without a recursion stack. It is essentially a flowchart [7].

Lambda-notation is used to specify and manipulate functional abstractions. The expression $\lambda xy.e$ denotes the function defined by expression e with formal parameters x and y. The form $\lambda(x, y).e$ is used when the formal parameter is a pair. The free identifiers in e are bound lexically, allowing one to build abstractions from the surrounding context.

A Model of Behavior

A digital circuit is a system of non-terminating behaviors, or *signals*, formally modeled as infinite sequences. Considered as a domain,

$$SIGNAL_V = V \times SIGNAL_V .$$

V is a domain of ground values. "$\langle v_0, \langle v_1, \langle v_2, \ldots \rangle\rangle\rangle$" is abbreviated to "$\langle v_0, v_1, v_2, \ldots \rangle$". Signals model the history of values on a conducting path. They are infinite because circuits do not terminate in the sense that programs do. A correct circuit passes through a predictable state but continues to operate thereafter. Signals are expressed by constants, simple terms, and initialized signals:

$$\boxed{v} = \langle v, \boxed{v} \rangle$$
$$\boxed{f}(S_1 \ldots S_n) = \langle f(S_1^\bullet \ldots S_n^\bullet),\ \boxed{f}(S_1^\dagger \ldots S_n^\dagger) \rangle$$
$$v\,!\,S = \langle v, S \rangle$$

where $f : V^n \to V$ is a strict operation on the ground domain; and if $S = \langle v_0, v_1, v_2, \ldots \rangle$, then $S^\bullet = v_0$ and $S^\dagger = \langle v_1, v_2, \ldots \rangle$.

The boxes distinguish the behavioral interpretation of the underlying symbols from their discrete interpretation in specifications. Although the discrimination is sometimes helpful, it is unnecessary when reasoning about terms. The elementary algebra for terms, based on composition, combination, construction, and substitution, commutes with the signal-interpretation. For example,

$$\boxed{f \circ g}(S) \equiv \boxed{f}(\boxed{g}(S)), \quad \text{and} \quad \boxed{\lambda(u,v).f(u,g(v,u))}(X,Y) \equiv \boxed{f}(X, \boxed{g}(Y,X)).$$

This transparency is the basis of the argument for using a functional calculus in digital engineering.

A *circuit description* is a set of mutual signal-defining equations, for example,

$$X = n\,!\ \boxed{decrement}(X)$$
$$Y = 1\,!\ \boxed{multiply}(X,Y)$$
$$P = \quad \boxed{zero?}(X)$$

This recursive system abbreviates the recurrence relation

$$\begin{array}{ll} X_0 = n & Y_0 = 1 \\ X_{k+1} = X_k - 1 & Y_{k+1} = X_k \times Y_k \end{array} \qquad P_k = (X_k = 0)$$

The model says nothing about time, but if it is assumed that values with the same index are coincidental, then the example describes a clocked sequential circuit, or *synchronous system* [4]. Initialized signals denote clocked storage elements; other terms are combinational. All feed-back loops pass through a storage element. There is an obvious schematic correspondence; signal identifiers name paths whose connectivity is prescribed by the equations:

a flowchart. Control hardware is systematically derived from the control description [4, 15]. Since iterative schema characterize "flowchartability", writing an iterative specification is equivalent to conceiving a sequential-control algorithm.

Non-iterative specifications represent a higher level of specification that is not available in the conventional methodologies. Their direct translation to circuits is not always possible; non-linear recursions must be removed or implemented. For both tasks, systematic formal techniques exist from research in program transformation, program synthesis, denotational semantics, and comparative schemata.

An Example

This section develops a small example of circuit-description synthesis and demonstrates various uses of functional abstraction in the derivation of a realization. An iterative version of an initial specification is obtained by implementing recursion with a stack. Since the initial specification is linear, a stack is not strictly necessary. However, its incorporation provides a context for a discussion of the treatment of complex data types. The initial steps of the derivation follow a standard pattern and are presented with little justification; more complex exercises can be found in [8] and [9], which cite related research. The interested reader might begin with [13] and [2] and find mathematical foundations in [7] and [11].

- Let us take as a specification the general linear recursion scheme

$$F(x) \Leftarrow p(x) \rightarrow f(x),\ g(x, F(h(x))).$$

The variables p, f, g, and h stand for arbitrary combinations of strict operations on a fixed but unspecified ground domain. The example is quite general: any linear recursion equation can be expressed in this form. (The *factorial* specification shown earlier is an instance of this scheme.)

- Introduce a continuation—an abstraction for control—to specify an order for the computation.

$$F(x, [\lambda v.v]) \text{ where}$$
$$F(x, \gamma) \Leftarrow p(x) \rightarrow \gamma f(x),\ F(h(x),\ [\lambda v.\gamma g(x, v)]).$$

Landin was the first to use functional abstraction to describe control in this way [5]. In denotational semantics, this is a standard step in the development of an operational semantics for a programming language [11].

- Choose a stack to represent the continuation and define a function to interpret that representation.

$$F(x, empty) \text{ where}$$
$$F(x, c) \Leftarrow p(x) \rightarrow G(f(x), c),\ F(h(x),\ push(x, c)).$$
$$G(v, c) \Leftarrow empty?(c) \rightarrow v,\ G(g(top(c), v),\ pop(c)).$$

(For the *factorial* instance of this specification, an accumulator is a better representation choice, giving rise to the iterative version of Fac shown earlier [13].)

- This version of F is iterative. Add a token encoding which of the functions F and G is "in control".

$$
\begin{array}{l}
F(\mathtt{F}, x, empty) \\
\quad where\ F(s, x, c) \ \Leftarrow \\
\qquad (s = \mathtt{F}) \rightarrow \\
\qquad\quad [p(x) \rightarrow F(\mathtt{G}, f(x), c),\ F(\mathtt{F}, h(x), push(x, c))] \\
\qquad (s = \mathtt{G}) \rightarrow \\
\qquad\quad [empty?(c) \rightarrow v,\ F(\mathtt{G}, g(top(c), x),\ pop(c))].
\end{array}
$$

- An instance of the desired scheme is obtained by distributing the conditional across the calls to F. The general distributive law is

$$p \rightarrow f(x),\ g(y) \quad \equiv \quad (p \rightarrow f, g)(p \rightarrow x, y)$$

First, define a combinator ("macro") for the selection operation:

$$sel \stackrel{\text{def}}{=} \lambda(q, r, v_0, v_1, v_2).\ q \rightarrow [r \rightarrow v_0, v_1], v_3$$

With some rearrangement and identification of terms, distributing *sel* yields

$$
\begin{array}{l}
F(s, x, c) \ \Leftarrow \\
\quad let\ q = equal?(s, \mathtt{F}) \\
\qquad\ r = p(x) \\
\qquad\ t = top(c) \\
\qquad\ e = empty?(c) \\
\quad in \\
\quad and(not(q), e) \rightarrow x, F(\ sel(q, r, \mathtt{G}, \mathtt{F}, \mathtt{G}), \\
\qquad\qquad\qquad\qquad\qquad sel(q, r, f(x), h(x), g(t, x)), \\
\qquad\qquad\qquad\qquad\qquad sel(q, r, c, push(x, c), pop(c))\ ).
\end{array}
$$

- This version of F is in the characteristic form and is realized by

$$
\begin{array}{rl}
F(s^0, x^0, c^0) \ \Leftarrow & (X,\ RDY)\ where \\
S = s^0\ !\ & \boxed{sel}(Q, R, \boxed{\mathtt{G}}, \boxed{\mathtt{F}}, \boxed{\mathtt{G}}) \\
X = x^0\ !\ & \boxed{sel}(Q, R, \boxed{f}(X), \boxed{h}(X), \boxed{g}(T, X)) \\
Q = & \boxed{equal?}(s, \boxed{\mathtt{F}}) \\
R = & \boxed{p}(X) \\
RDY = & \boxed{and}(\ \boxed{not}(Q), E) \\
T = & \boxed{top}(C) \\
E = & \boxed{empty?}(C) \\
C = c^0\ !\ & \boxed{sel}(Q, R, C, \boxed{push}(X, C), \boxed{pop}(C))
\end{array}
$$

Abstract Components

Signal C in the derived circuit description ranges over stacks and, therefore, could hardly be regarded as a true data path. The factorization below yields a circuit over more concrete signals by introducing an information-hiding component analogous to an abstract data type. We continue the example, now concentrating on the complex signals

$$\begin{aligned} T &= \boxed{top}(C) \\ E &= \boxed{empty?}(C) \\ C &= c^0 \,!\, \boxed{sel}(Q, R, C, \boxed{push}(X, C), \boxed{pop}(C)) \end{aligned}$$

The goal is to replace the signal C with an *abstract component* which manages the stack. At the same time, the surrounding circuit should retain the selector that determines what is done to the stack. The key step is to introduce a "signal" that ranges over operations and is applied to the appropriate operands.

$$\begin{aligned} C &\simeq c^0 \,!\, \boxed{I}(X, C) \\ \boxed{I} &\simeq \boxed{sel}(Q, R, \boxed{\lambda(v,c).c}, \boxed{\lambda(v,c).push(v,c)}, \boxed{\lambda(v,c).pop(c)}) \end{aligned}$$

This gets the selection out of C's definition, but signals don't range over operations. $\boxed{I}$ is really a signal of instructions. Define an interpretation for instruction-representations.

$$\begin{aligned} interpret \stackrel{\text{def}}{=} \lambda(i, x, c)\,.\, &(i = \texttt{STET}) \rightarrow c, \\ &(i = \texttt{PUSH}) \rightarrow push(x, c), \\ &(i = \texttt{POP}) \rightarrow pop(c). \end{aligned}$$

Instruction-signal I becomes

$$I = \boxed{sel}(Q, \boxed{\texttt{STET}}, \boxed{\texttt{PUSH}}, \boxed{\texttt{POP}})$$

Now define the abstract component $STACK$ to be

$$\begin{aligned} &STACK\ (c^0, I, X) \ \Leftarrow\ (\boxed{empty?}(C), \boxed{top}(C)) \\ &\quad where \\ &\quad C = c^0 \,!\, \boxed{interpret}(I, X, C) \end{aligned}$$

With the incorporation of $STACK$, the circuit description is now in terms of concrete signals, schematically depicted below.

$$\begin{aligned} F(s^0, x^0, c^0) \ &\Leftarrow\ (X,\ RDY)\ where \\ S &= s^0 \,!\, \boxed{sel}(Q, R, \boxed{\texttt{G}}, \boxed{\texttt{F}}, \boxed{\texttt{G}}) \\ X &= x^0 \,!\, \boxed{sel}(Q, R, \boxed{f}(X), \boxed{h}(X), \boxed{g}(T, X)) \\ Q &= \boxed{equal?}(s, \boxed{\texttt{F}}) \\ R &= \boxed{p}(X) \\ RDY &= \boxed{and}(\boxed{not}(Q), E) \\ I &= \boxed{sel}(Q, R, \boxed{\texttt{STET}}, \boxed{\texttt{PUSH}}, \boxed{\texttt{POP}}) \\ (E, T) &= STACK(c^0, I, X) \end{aligned}$$

Define a *component* to be a signal-of-operations and extend the definition of application. Informally,

$$\boxed{f} = \langle f, \boxed{f} \rangle$$

$$C(X_1 \ldots X_n) = \langle C^\circ(X_1^\circ \ldots X_n^\circ),\ C^\dagger(X_1^\dagger \ldots X_n^\dagger) \rangle$$

This model supports the steps leading to the definition of *STACK* in the example, but also raises problems in the treatment of multiple-valued or partial components and in the assurance of a uniform signature. In practice, these problems are avoided by constraining the way in which components are introduced, just as combinational feed-back is avoided by the transformation process. The component $\boxed{I}$ was composed of strict functions on a normalized parameter.

A derived circuit description is still subject to refinement. Transformations on signals, such as those developed by Sheeran [10], seem appropriate at this level, as does reasoning based on interval logic. The theorem giving a characteristic specification for synchronous systems establishes the temporal assertion, *"ANS is* $F(v_1, \ldots, v_n)$ *as soon as RDY is true."* An algebra for local refinement should maintain this correctness property. It is reasonable to assume that some abstract-component *implementations* come with their own clocks. *STACK* was incorporated as a synchronous sub-circuit. To be useful in practice, the technique must be extended to cope with the implied communication protocols.

Circuit descriptions are directly interpretable as programs, with signals being represented as recursively defined lists. The resulting logical simulation is a data structure that can be manipulated with an editor. Figure 1, which shows an editor for the *factorial*-circuit written in Daisy, illustrates how little need be involved in simulating logical behavior. This kind of experimentation can reveal unforeseen properties of behavior. In the example, $n!$ appears one cycle before *RDY* is asserted, suggesting the addition of a $\boxed{zero?}$-test to the output of $\boxed{decrement}$. Signal Y eventually becomes stable, a fact which may be useful in designing an interface to the circuit.

Acknowledgments

This material is based on work supported by the National Science Foundation under grant number DCR84-05241. John O'Donnell contributed key insights, which I hope are adequately reflected here. I am indebted to Franklin Prosser and David Winkel for their guidance and encouragement.

References

[1] Abelson, H. and Sussman, G. J., *Structure and Interpretation of Computer Programs* (The MIT Press and McGraw-Hill, Cambridge, 1985).

[2] Darlington, J. and Burstall, R. M., A system which automatically improves programs, *Acta Informatica*, 6 (1976) 41–60.

[3] Henderson, P., *Functional Programming: Application and Implementation* (Prentice-Hall, Englewood Cliffs, 1980).

[4] Hill, F. J. and Peterson G. R., *Introduction to Switching Theory and Logical Design* (Third Ed.) (John Wiley&Sons, New York, 1981).

[5] P. J. Landin, An abstract machine for designers of computing languages, *Proc. IFIP Congress 65*, Vol. 2, (Spartan Books, Washington; MacMillan, London, 1966)

[6] McCarthy, J., A basis for a mathematical theory of computation, in: P. Braffort and D. Hirschberg (eds.), *Computer Programming and Formal Systems* (North-Holland, Amsterdam, 1963).

[7] Manna, Z., *Mathematical Theory of Computation,* (McGraw-Hill, New York, 1974).

[9] Johnson, S. D., Applicative Programming and Digital Design, *Eleventh Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1984).

[9] Johnson, S. D., *Synthesis of Digital Designs from Recursion Equations, The ACM Distinguished Dissertation Series,* (The MIT Press, Cambridge, 1984).

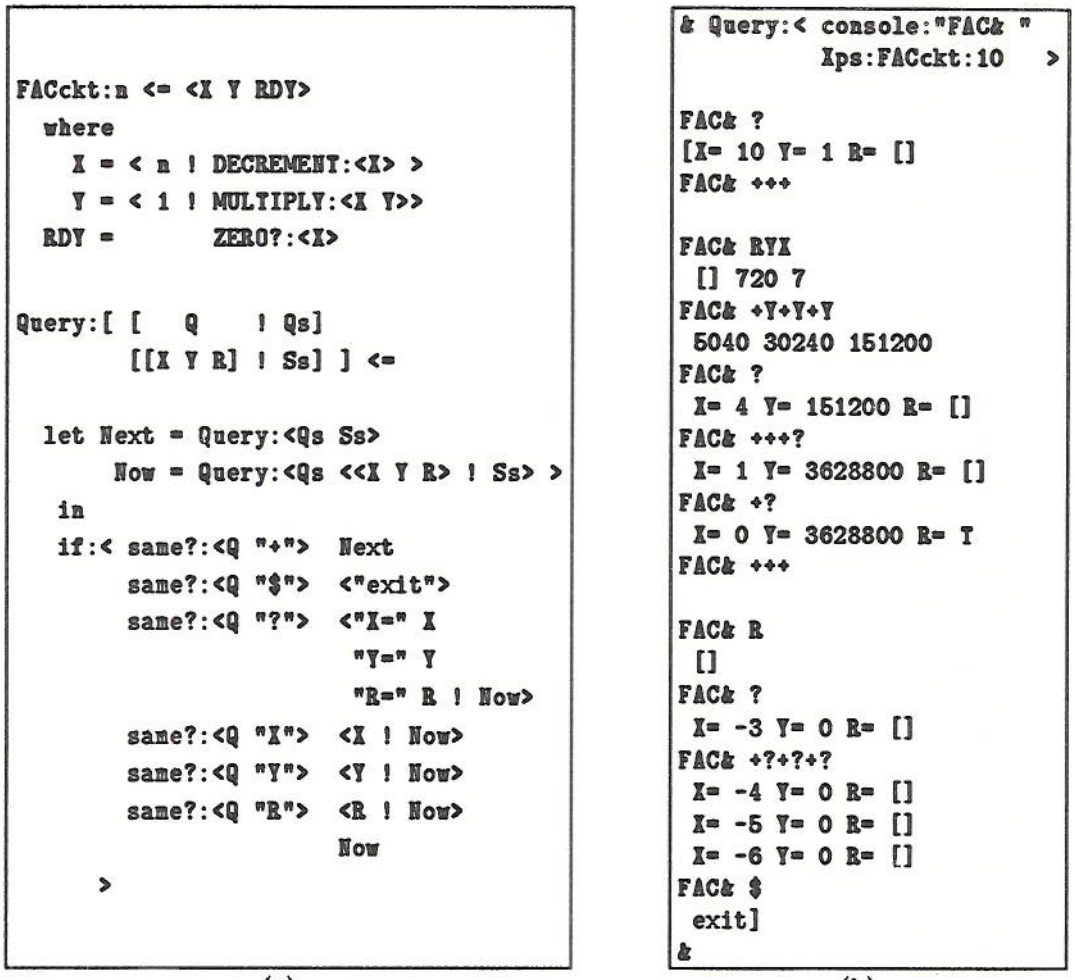
[10] Sheeran M., Designing regular array architectures using higher order functions, in: Jouannaud, J-P (ed.), *Functional Programming Languages and Computer Architecture* (Springer [*LNCS* No. 201], Berlin, 1985).

[11] Stoy, J. E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory* (The MIT Press, Cambridge, 1977).

[13] Wand M., Continuation based program transformation strategies, *J. ACM* **27**(1) (1980) 164–180.

[14] Wand, M., Deriving target code as a representation of continuation semantics, *ACM Trans. on Programming Languages and Systems* **4** (1982) 496–517.

[15] Winkel D.E., and Prosser F.P., *The Art of Digital Design* (Prentice-Hall, Englewood Cliffs, 1980).



```
FACckt:n <= <X Y RDY>
  where
    X = < n ! DECREMENT:<X> >
    Y = < 1 ! MULTIPLY:<X Y>>
  RDY =       ZERO?:<X>

Query:[ [   Q    ! Qs]
        [[X Y R] ! Ss] ] <=

  let Next = Query:<Qs Ss>
      Now = Query:<Qs <<X Y R> ! Ss> >
   in
   if:< same?:<Q "+">  Next
        same?:<Q "$">  <"exit">
        same?:<Q "?">  <"X=" X
                        "Y=" Y
                        "R=" R ! Now>
        same?:<Q "X">  <X ! Now>
        same?:<Q "Y">  <Y ! Now>
        same?:<Q "R">  <R ! Now>
                       Now
      >
```

(a)

```
& Query:< console:"FAC& "
          Xps:FACckt:10   >

FAC& ?
[X= 10 Y= 1 R= []
FAC& +++

FAC& RYX
 [] 720 7
FAC& +Y+Y+Y
 5040 30240 151200
FAC& ?
 X= 4 Y= 151200 R= []
FAC& +++?
 X= 1 Y= 3628800 R= []
FAC& +?
 X= 0 Y= 3628800 R= T
FAC& +++

FAC& R
 []
FAC& ?
 X= -3 Y= 0 R= []
FAC& +?+?+?
 X= -4 Y= 0 R= []
 X= -5 Y= 0 R= []
 X= -6 Y= 0 R= []
FAC& $
 exit]
&
```

(b)

Figure 1
Daisy Experiment with the Factorial Circuit Description

The colon denotes application; `< ⋆...⋆ ! ⋆ >` *builds lists; '&' is Daisy's prompt. The primitive* `console`*, whose argument is a prompt, produces a character-stream from the keyboard;* `same?` *compares values;* `if` *is like Lisp's* `COND`*.*

`DECREMENT`*,* `MULTIPLY`*, and* `ZERO?` *are arithmetic operations extended to "streams" (lists), and* `Xps` *transposes a list-of-streams to a stream-of-lists. Each is trivial to define owing to Daisy's interpretation of list-application [9].*

`FACckt` *models the factorial-circuit.* `Query` *is a stream-oriented interpreter for questions about* `FACckt`*'s behavior. The source (a) has been modified; Daisy's syntax is slightly more crude. The transcript (b) is verbatim.*

decrement
1
!
X
zero?
P
multiply
1
!
Y