PROTOTYPING DATA FLOW BY TRANSLATION INTO SCHEME

Pee-Hong Chen
Daniel P. Friedman
Computer Science Department
Lindley Hall 101
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 147

PROTOTYPING DATA FLOW BY TRANSLATION INTO SCHEME

by

Pee-Hong Chen
Daniel P. Friedman

August, 1983

Research reported herein was supported in part by the National Science Foundation under grant numbers MCS 79-04183, MCS 82-03978, and MCS 83-04567.

PROTOTYPING DATA FLOW BY TRANSLATION INTO SCHEME

Pee-Hong Chen

Daniel P. Friedman

Computer Science Department
Lindley Hall 101
Indiana University
Bloomington, IN 47405

Abstract

We consider language modeling using Scheme, an applicative order, lexically scoped dialect of Lisp. In Lisp, programs are represented by list structures. By representing programs of other languages as list structures with carefully placed keywords it is possible to define the keywords as either functions or syntactic extensions. Proper consideration of the lexical scoping yields even more control than would be expected. As an example of this approach we prototype Data Flow, a concurrent computation system, by expanding each actor or arc into a function invocation and by expanding a network into a Scheme function that is compiled and executed. This approach avoids the familiar interpretation step used in most prototyping approaches, yielding improved performance.

Research reported herein was supported in part by the National Science Foundation under grant numbers MCS 79-04183, MCS 82-03978, and MCS 83-04567.

§1 Introduction

The central idea of Data Flow is to express computations through successive transformations of the token values that mark a graph structure. It is due to the inherent nature of the graph structure that highly parallel computations can be achieved. Various node types are available for the system. Synchronization inside the system is realized by protocols (or firing policies, in Data Flow jargon) that decide the passing (flowing) of tokens (data) between any two nodes. In a Data Flow network all nodes fire concurrently.

To simulate Data Flow, we use Scheme[1] as the base language upon which a Data Flow program translator is defined. Scheme has the power to expand syntactic expressions into executable code. In addition, it supports some parallel processing primitives. Taking advantage of these features, our mechanism is able to convert Data Flow networks into stand-alone Scheme object programs. When an object program is executed, it spawns several processes running concurrently, each simulating a Data Flow node.

The metaphor derived from our work is that, given a suitable language syntax, Scheme processes programs of that language in a direct and natural way [Cl83]. Conventionally, to model a system is to write a language simulator for it so that the data (programs written in the target language) must be interpreted step by step during execution. In our mechanism, however, the data become executable programs. Our facility is more like a naive compiler (preprocessor) than like an interpreter. Consequently, we start with a brief review of Scheme in the next section. The basic concepts of Data Flow and its architecture under our base language are then introduced. In the section following we develop the implementation for our target system. Finally, some anticipated results of this work and research problems are mentioned in the conclusion.

§2 Scheme : An Overview

The programming language Scheme was designed and first implemented at MIT in 1975 by Gerald J. Sussman and Guy L. Steele, Jr. It is based upon the lambda calculus described by Alonzo Church [Ch41] and serves as "a simple concrete experimental domain for certain issues of programming semantics and style". The revised report on Scheme was published by Steele and Sussman in 1978 [St78]. In 1980 they implemented their first Scheme VLSI chip and at the same time a full report on their work was released [St80].

Scheme is a dialect of Lisp with features including applicative order, lexical scoping, proper tail recursion, and block structure. Most important of all, Scheme — unlike most Lisp dialects — treats functions and continuations as first class objects. The Scheme used in our work is Scheme 84 [Fr84] which is written in Franz Lisp [Fr84].

The syntax of the kernel of Scheme follows,

---

[1] on a VAX 11/780 running under Berkeley Unix. Unix is a trademark of Bell Laboratories.

```
expression  ::=   constant
                | identifier
                | ( if expression expression expression )
                | ( lambda ( {identifier} ) expression )
                | ( set! identifier expression )
                | application
                | syntactic-extension
application  ::=  ( {expression} )
syntactic-extension  ::=  ( keyword {object} )
```

A number of traditional syntactic extensions such as **list**, **cond**, **case**, **begin**, **let**, and **letrec** [La 65] are provided to enhance the Scheme system. User definition of syntactic extensions is possible. For example, **while** is defined by the following syntactic extension

```
(while bool body) ≡

(letrec ([*loop (lambda ()
                  (if bool
                      (begin body (*loop))
                      t))])
        (*loop))
```

The effect of normal order evaluation can be obtained when necessary by using *thunks*, functions of no arguments. We use the syntactic extensions

`(freeze exp)` ≡ `(lambda () exp)` *; this creates a function of 0 arguments*

and

`(thaw th)` ≡ `(th)` *; this applies the function* `th` *to a list of 0 arguments.*

To control parallelism, we use the functions **fork** and **block**. These functions are not primitives, but are defined in terms of a more primitive process-control mechanism [Fr84]. The function **fork** is used to initiate parallelism. It takes a thunk, starts it running (thaws it) as a parallel process **P**. The function **block** is a thunk that kills any process that invokes it, leaving its subprocesses uneffected.

The idea of *test-and-set* is implemented via a **box**, a cons-cell initialized as `(nil)`. To manipulate a box, we have **test-and-set-box!** and **clear-box!**. The **test-and-set-box!** tests a box and returns `t` if the box is set, otherwise it sets the box (changes the box to `(t)`) and returns `nil`. The function **clear-box!** is used to reset a box.

The concept of data abstraction is fundamental to the way we write code in Scheme. For example, the basic operations regarding the use of a box can be abstracted by the following object:

```
(define bi-sem
   (lambda ()
      (let ([b (cons nil nil)])
         (rec
            self (lambda (msg)
                   (case msg
                      [wait (while (test-and-set-box! b) (no-op))]
                      [signal (begin (clear-box! b) self)]
                      [otherwise (writeln msg "Unknown box operation!")])))))))
```

where a box here is implemented as a binary semaphore and the pair of **wait** and **signal** forms a *region* by busy waiting. The object **bi-sem** is a thunk. It creates an environment local to the object only when it is thawed. Internal data structures as well as the related manipulation operations are hidden. When an identifier is declared as a box (the thawing of **bi-sem**), it gets bound to a one argument function that when given the message **'wait** tests and sets the box, or when given the message **'signal** resets the box. Since b is local to the function it can only be accessed by sending the function **'wait** or **'signal** messages [Mo73].

We now present another example of an abstract data type in Scheme. It is an abstraction for the construction and manipulation of a unit buffer, which will be the basic data structure in our Data Flow System (Sec. 3.4).

```
(define unit-buffer
   (lambda ()
      (let ([c 'no-contents])
         (rec
            self (lambda (msg)
                   (case msg
                      [send (lambda (x)
                               (begin
                                  (while (not (eq? 'no-contents c))(no-op))
                                  (set! c x)
                                  self))]
                      [receive (begin
                                  (while (eq? 'no-contents c) (no-op))
                                  (begin0 c (set! c 'no-contents)))]
                      [otherwise (writeln msg "Unknown buffer operation!")])))))))
```

where **begin0** takes two arguments, executes them in order, and returns the first.

When an identifier, say **x**, is bound to **(unit-buffer)**, a register containing the string **no-contents** (the initial buffer content) is created and **self** is returned. This data structure is accessible only to the set of local operations defined in the **self** body. For example, **x** takes its argument using **send** Curried, so **((x 'send) 3)** sends the value **3** to buffer **x**.

The operations **send** and **receive** are similar to write and read on a register. However, there are conditions. **Send** must wait until the register's content has been read. **Receive** must wait until the register has content to read and after reading, **receive** is obliged to erase the content making a future **send** possible.

3.1 What Is Data Flow?

The essence of Data Flow was first introduced in the IBM 360/91 system in the 1960's [IB67]. Early work on Data Flow models can also be traced back to R. Karp and R. Miller's research in 1966 [Ka66]. Since then Data Flow has been extensively studied by many researchers as an alternative to von Neumann architecture. Today, there are several versions of Data Flow systems, all somewhat similar, and all unified by graph structured computation.

A Data Flow language is any functional language based entirely upon the notion of data flowing from one functional unit to another. It is this concept of *flow* that provides Data Flow languages with the power to represent program definitions as graphs. The implicit advantages of applicative programs, such as modularity and concurrency, can be seen even more transparently from the corresponding program graphs.

Data in Data Flow program graphs is always viewed as flowing on arcs from one node to another in a stream of discrete *tokens*. Hence tokens are the instantaneous descriptions of data objects. Each node in the system repeatedly takes tokens arriving at its input arcs (if any) and produces tokens at its output arcs. The transformation of the values of tokens for a node is decided by a firing policy. The nodes in a graph are called *actors*. Collectively, an actor and its input/output arcs form a *node* in a more generalized sense. Different types of actors are associated with different firing policies.

Dennis proposed a Data Flow model [De74] upon which we base our system. In his model, a *link* has an in-degree of exactly one while it must have an out-degree of at least one. That is, a link can have many output arcs, but only one input arc. The capacity of any arc is restricted to one. Synchronization among all tokens of the same data set is achieved as a natural consequence of this restriction.

Most Data Flow nodes cannot fire unless their input arcs have tokens, but there are some exceptions (e.g. selectors). In addition, since only a single token can rest on any arc at any time, a Data Flow node cannot fire until its output arcs are free of tokens. So when firing a Data Flow node, the input tokens are, in general, consumed by the node and if any of its output arcs is occupied by the previous token, the node puts the current output token on that arc as soon as it becomes free. There is extensive literature on Data Flow. (See for example [Da82, Tr82].)

3.2 Data Flow Actors

There are five elementary actors: *constants, operators, gates, conditional constructors,* and *I/O's*. Operators can be further categorized into *arithmetic operators, boolean operators,* and *general predicates (deciders)*. For gates, we have the *True gates (T-gates)* and the *False gates (F-gates)*. For conditional constructors, we have *selectors* and *distributors*. Together, gates and conditional constructors are sometimes called *regulators*. In addition, there are *writers* and *readers* for I/O actors.

To see how to use a language like this in defining programs, a conditional expression in Data Flow is depicted by Figure 1. Let us now focus on one execution cycle of that expression. Upon receiving input data given by the user, the link (node 7) broadcasts **b** to each of its output arcs. Depending on the Boolean value of this control token, either node 3 or node 4, the two gates, will be able to pass either **x** or **y** to one of the entries of the selector (node 5). The gate bearing an attribute that does not match the control token value simply consumes the user input and passes nothing over. The selector

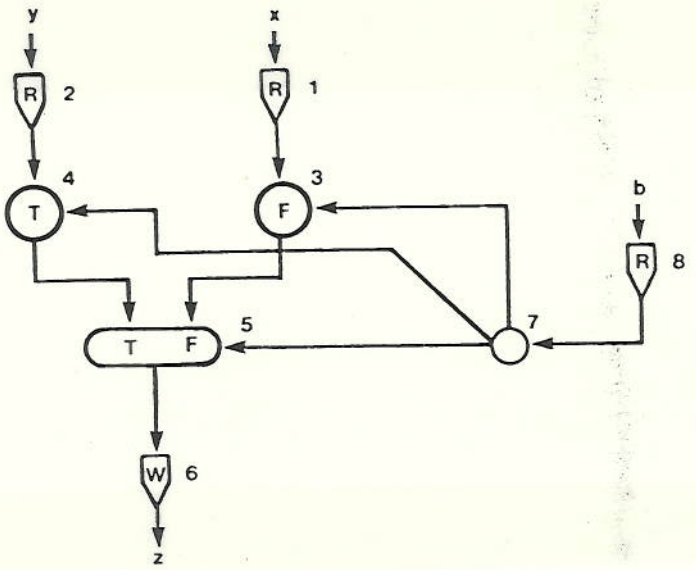

Figure 1. A conditional expression: $z =$ **if** b **then** x **else** y

then propagates the token from the matching entry to its exit. Finally, the writer grabs that token and prints it on the screen.

3.3 Firing Policies

Without loss of generality, we assume the `link` as an additional actor type. Data Flow actors' firing policies are stated below.

1. **Links** : Firing a link is simple. It consumes its input token and then sends a copy of that token to each of its output arcs.
2. **Constants** : Constant actors have no input arcs. A constant generates its value as output and is firable whenever its output arc is empty.
3. **Operators** : An arithmetic operator computes primitive functions, such as $+$ and $\sqrt{\ }$. A decider is the actor for primitive predicates, such as $=$ and $>$. Boolean operators are the boolean functions of their control-valued inputs, such as $\wedge$, $\vee$ and $\neg$. An operator fires when its input arcs have tokens and its output arc is empty.
4. **Gates** : There are two types of gates: T-gates and F-gates. A gate has two inputs, the control entry and the data entry. A T-gate can propagate its input data token to its output arc if its input control token is **true**, otherwise the gate consumes the data token and produces no output. A similar situation applies to F-gates.
5. **Conditional Constructors** : Like gates, conditional constructors allow control tokens to regulate the flow of data tokens. A selector has three inputs, a control entry and two data entries, **true** and **false**. In firing, it consumes the data token from the data entry with type matching the value of control token and leaves the other data entry unaffected. This is the only case an actor can fire in the absence of some of its input tokens. A distributor looks almost identical to a selector except the directions of the flow of data are reversed. In the case of firing a distributor, a token is absorbed from the data input and passed to one of the outputs whose type matches the value of the control token.

6. **I/O's** : I/O actors act like links except that readers receive tokens (input data) from the user and writers send tokens (results of computation) to the user.

3.4 A Data Flow Language

Key issues for simulating Data Flow networks include:

— *All active actors fire concurrently.*

— *All actors are active as soon as the program is activated and will stay active until the user decides to abort the program execution.*

— *Different actors have different firing rules.*

To model the system, we want to create a mechanism that transforms the data (a Data Flow program graph) into an executable (compiled) Scheme function instead of some intermediate form that needs to be interpreted. The first step is to define a language describing Data Flow networks. We present the following extended BNF syntax:

```
program  ::=  (define-network pgm-name ( {arc} ) ( {node} ))

   arc   ::=  name-of-arc | [ name-of-arc any-Scheme-constant ]
name-of-arc ::= any-Scheme-atom

  node   ::=    [ link in ( list {out} ) ]
              | [ constant any-Scheme-constant out ]
              | [ operator any-Scheme-function ( list {in} ) out ]
              | [ T-gate C-in D-in out ]
              | [ F-gate C-in D-in out ]
              | [ selector C-in T-in F-in out ]
              | [ distributor C-in D-in T-out F-out ]
              | [ reader any-Scheme-string out ]
              | [ writer any-Scheme-string in ]

in | C-in | D-in | T-in | F-in | out | T-out | F-out ::= name-of-arc
```

Given the syntax above, we can use the Scheme syntactic extension facility to generate a Scheme function definition which can be directly executed. In particular, `define-network` is the following syntactic extension:

```
(define-network pgm-name (arc1 arc2 ... arcm) (node1 node2 ... noden)) ≡

(define pgm-name
   (lambda (killall delay-time)
      (let ([arc1 **processed-form-for-arc1**]
            [arc2 **processed-form-for-arc2**]
            ...
            [arcm **processed-form-for-arcm**])
           (begin (define MUTEX (bi-sem))
                  node1
                  node2
                  ...
                  noden
                  (supervisor killall delay-time))))
```

The notion ****processed-form-for-arci**** represents the initial state for **arci**. It can be the thawing of **unit-buffer** if **arci** is just a name, or it can be the thawing of **unit-buffer** followed by a **send** operation if the arc is a name-value pair.

A direct wrapping with no preprocessing of the nodes into the program's object code is shown in the above expansion. This follows because the concrete syntax for nodes has been arranged to be executable Scheme code. This code, when executed will initialize a Scheme process which simulates the node. Thus, when the function **pgm-name** is applied to its arguments, each of these processes is started. They will then execute in parallel, sharing the semaphore **MUTEX**, thus simulating the entire Data Flow network.

Any **node** is a list headed by one of the keywords **link**, **operator**, **constant**, **selector**, **distributor**, **T-gate**, **F-gate**, **reader**, and **writer**. These words are plain syntax from the programmer's point of view. But they become the functions used to initiate parallel actor processes when **pgm-name** is executed.

§4 Prototyping Data Flow

4.1 Program Execution

To execute a Data Flow program, we first initiate its corresponding Scheme function with a killing routine and a delay time as arguments. The following code illustrates this.

```
(define run
   (lambda (pgm time)
      (letrec ([kill (fork (freeze (pgm killall time)))]
               [killall (freeze
                          (begin
                             (writeln "Program aborted upon user request.")
                             (kill t)))])
              (vanish))))
```

Recall that the object function of a Data Flow program is a lambda expression of two arguments. When **run** is invoked, we initiate the program with the arguments **killall** and **time** by the **fork** facility which in turn gives back an object bound to the identifier

`kill`. The routine `killall` is a frozen object that when thawed (by the supervisor) invokes `kill` with the parameter `t` aborting all processes. It is important that `run` vanishes after `kill` and `killall` are defined.

When a program starts execution, the killing routine `killall` and a delay time specified by the user at Data Flow top level are passed to the supervisor process generated by `define-network` which runs in parallel with all actor processes. The supervisor is the only process other than I/O actors that interacts with the user during program execution. Like readers, it prompts the user periodically asking just one question: "Aborting execution?". The frequency of this inquiry is determined by the delay time. If user response is negative, it does nothing. However, if the response is positive, it thaws `killall` and thus terminates the program.

4.2 Firing Actors

In our implementation, each node is realized as a Scheme function that when called initiates (forks) an infinite process in the background. In general, an actor looks like:

```
(define some-actor
   (lambda (...)
      (letrec ([fire (freeze (begin ... (thaw fire)))])
              (fork fire))))
```

where the first ellipsis refers to the actor's required arguments and the second ellipsis refers to the realization of its firing policy.

We provide an `actor` creator with the syntactic extension

```
(actor formal-parameters firing-policy) ≡

(lambda formal-parameters
   (letrec ([fire (freeze (begin firing-policy (thaw fire)))])
           (fork fire)))
```

We create an actor by invoking `actor` with a parameter list and a firing policy. The identifiers defined in `actor` are nonlocal to the policy. Thus, the code in a firing-policy can access its formal parameters.

The following are implementations of the actor types.

1. Links: A link has only one input arc. In firing, it receives an input token from that arc and broadcasts the token to each of its ouput arcs. We have[4]

```
(define link
   (actor (in out-arcs)
      (let ([token (in 'receive)])
         (mapc (lambda (arc) ((arc 'send) token)) out-arcs))))
```

2. Constants : A constant has no input arcs. It fires (sends out its constant value) whenever its output arc is free. That means we can define a constant as

```
(define constant
   (actor (val out)
      ((out 'send) val)))
```

3. Operators : The number of input arcs for an operator depends on the arity of its underlying Scheme function. For uniformity, we assume that the actor takes a list of input arcs. In firing, it receives all input tokens from the arcs, applies the function to the resulting list of tokens, and sends a copy of the result to its output arc.[4]

```
(define operator
   (actor (function in-arcs out)
      ((out 'send) (apply function
                          (mapcar (lambda (arc) (arc 'receive)) in-arcs)))))
```

4. Gates : There are two types of gates, T-gates and F-gates. In firing, the data input will be propagated to the output only if the control token value agrees with the gate attribute. We have

```
(define T-gate
   (actor (C-in D-in out)
      (let ([data (D-in 'receive)])
         (if (C-in 'receive) ((out 'send) data)))))

(define F-gate
   (actor (C-in D-in out)
      (let ([data (D-in 'receive)])
         (if (C-in 'receive) (no-op) ((out 'send) data)))))
```

5. Conditional Constructors :

(1) Selectors : A selector takes three input arcs: control, T, and F. If the control entry has a value of **true** , then it propagates the T-entry token to its output arc, else it propagates the F-entry value to the output arc.

---

[4] mapc in link and mapcar in operator could be parallel operations. For clarity, we have chosen the two to be sequential.

```
(define selector
   (actor (C-in T-in F-in out)
      ((out 'send) ((if (C-in 'receive) T-in F-in) 'receive))))
```

(2) Distributors : A distributor takes two input arcs: control and data entries, and two output arcs: T- and F-exits. If the control token is true, the data token is sent to the T-exit, otherwise it is sent to the F-exit. We can define the distributor as:

```
(define distributor
   (actor (C-in D-in T-out F-out)
      (((if (C-in 'receive) T-out F-out) 'send) (D-in 'receive))))
```

6. I/O's : The original Data Flow model does not have any I/O actors. To do software simulation, however, such an actor type is an inevitable extension. The system, through these I/O actors, is able to prompt the user asking for input data or responding with computed results. The user, also through these actors, is able to type in input data, to decide when to quit (using the keyword **eof** at any input prompt), or to monitor the results.

I/O actors are special: they become active as soon as the system is initiated like all other actors, but, exclusively, they also are involved in competition for the unique I/O device (the terminal). To mutually exclude the usage of this device, a semaphore is needed. We call this semaphore **MUTEX**, and it must be declared globally. In short, any operation related to the user interface (reader, writer, or supervisor) must be included within a **MUTEX** region. We have:

(1) Readers : A reader can be defined as follows.

```
(define reader
   (actor (prompt out)
      ((out 'send)
       (begin (MUTEX 'wait)
              (princ prompt)
              (let ([data (read)])
                   (begin (MUTEX 'signal)
                          (if (eq data 'eof) (vanish) data)))))))
```

A reader asks for input data from the user, and upon receiving the keyword **eof**, it **vanishes**, thereby bringing the reader out of the **MUTEX** competition.

(2) Writers : A writer simply receives the answer from its input arc and prints it on the screen. We can define:

```
(define writer
   (actor (prompt in)
      (let ([ans (in 'receive)])
           (begin (MUTEX 'wait) (writeln prompt ans) (MUTEX 'signal)))))
```

(3) Supervisor: The supervisor is a competitor for the terminal because it interacts with the user. We mentioned before that there can be deliberate bias against it in the **MUTEX** competition. The reason for creating bias is that frequent supervisor inquiries are irritating. As a solution, the user specifies a desired inquiry frequency when his program is initiated.

```
(define supervisor
   (actor (killall delay-time)
      (begin (delay delay-time)
             (MUTEX 'wait)
             (princ "Aborting execution? (y/n)  ")
             (let ([abort (eq (read) 'y)])
                  (begin (MUTEX 'signal)
                         (if abort (thaw killall)))))))
```

The supervisor is a pesudo actor (because it does not fire). The unfairness is realized by a simple delay mechanism. It aborts the whole system by invoking **killall** if the user's response to its inquiry is positive.

4.3 Two Sample Programs

We now introduce two examples which demonstrate our approach.

1. IF expression : We can program the conditional expression **if-network** (Figure 1) as follows.

```
(define-network if-network
     (A-1-3 A-2-4 A-3-5t A-4-5f A-5-6 A-7-3r A-7-4r A-7-5r A-8-7)
     ([reader   "x = " A-1-3]
      [reader   "y = " A-2-4]
      [T-gate   A-7-3r A-1-3 A-3-5t]
      [F-gate   A-7-4r A-2-4 A-4-5f]
      [selector A-7-5r A-3-5t A-4-5f A-5-6]
      [writer   "z = " A-5-6]
      [link     A-8-7 (list A-7-3r A-7-4r A-7-5r)]
      [reader   "b = " A-8-7]))
```

which expands to the following Scheme code

```
(define if-network
   (lambda (killall delay-time)
      (let ([A-1-3  (unit-buffer)] [A-2-4  (unit-buffer)] [A-3-5t (unit-buffer)]
            [A-4-5f (unit-buffer)] [A-5-6  (unit-buffer)] [A-7-3r (unit-buffer)]
            [A-7-4r (unit-buffer)] [A-7-5r (unit-buffer)] [A-8-7 (unit-buffer)])
           (begin (define MUTEX (bi-sem))
                  [reader "x = " A-1-3]
                  [reader "y = " A-2-4]
                  [T-gate A-7-3r A-1-3 A-3-5t]
                  [F-gate A-7-4r A-2-4 A-4-5f]
                  [selector A-7-5r A-3-5t A-4-5f A-5-6]
                  [writer "z = " A-5-6]
                  [link A-8-7 (list A-7-3r A-7-4r A-7-5r)]
                  [reader "b = " A-8-7]
                  (supervisor killall delay-time)))))
```

The user invokes (**run if-network** *n*) where *n* is a delay constant.

2. Exponentiation : Figure 2 illustrates an exponentiation function in Data Flow. Simple analysis shows that the graph can roughly be divided into three functional units: the left which keeps the value of *x* iterating, the center which keeps the current

value of z updated, and the right which counts down the value of *n*. The importance of data feedback technique in Data Flow programming can be summarized by this example. Our network description is

```
(define-network exp-network
    ([A-17-1r nil] [A-17-2r nil] [A-17-3r nil]
     A-4-2f    A-6-5   A-16-6   A-17-16r  A-12-5   A-13-18  A-17-12r
     A-17-13r  A-14-7  A-17-14r A-15-17   A-10-12  A-10-13  A-11-14
     A-11-15   A-1-16  A-2-10   A-3-11    A-8-3f   A-7-3t   A-5-2t
     A-6-1t    A-9-1f)
    ([selector A-17-1r A-6-1t A-9-1f A-1-16]
     [selector A-17-2r A-5-2t A-4-2f A-2-10]
     [selector A-17-3r A-7-3t A-8-3f A-3-11]
     [constant 1 A-4-2f]
     [operator * (list A-6-5 A-12-5) A-5-2t]
     [link     A-16-6 (list A-6-5 A-6-1t)]
     [operator 1- (list A-14-7) A-7-3t]
     [reader   'n = ' A-8-3f]
     [reader   'x = ' A-9-1f]
     [link     A-2-10 (list A-10-12 A-10-13)]
     [link     A-3-11 (list A-11-14 A-11-15)]
     [T-gate   A-17-12r A-10-12 A-12-5]
     [F-gate   A-17-13r A-10-13 A-13-18]
     [T-gate   A-17-14r A-11-14 A-14-7]
     [operator (lambda (n) (> n 0)) (list A-11-15) A-15-17]
     [T-gate   A-17-16r A-1-16 A-16-6]
     [link     A-15-17 (list A-17-16r A-17-1r A-17-12r A-17-13r
                             A-17-2r A-17-14r A-17-3r)]
     [writer    'z = ' A-13-18]))
```

The expansion of this network is similar to that of **if-network**. However, as shown in Figure 2, the major difference between the two expansions is that **exp-network** has three arcs initialized to **false** (i.e. **nil**). The body of the object function of **exp-network** is

```
(let ([A-17-1r (((unit-buffer) 'send) nil)]
      [A-17-2r (((unit-buffer) 'send) nil)]
      [A-17-3r (((unit-buffer) 'send) nil)]
      ...)
     (begin (define MUTEX (bi-sem))
            ...
            (supervisor killall delay-time)))
```

where the first ellipsis refers to the rest of the arcs to be declared and the second one refers to the sequence of wrapped nodes corresponding to the **nodes** declared above.

§5 Conclusions

We have seen how a highly parallel system like Data Flow can be easily prototyped by translation into Scheme. In fact, separate projects on modeling languages for coordinated computing [Fi83] such as Modula II [Wi82], Cell [Si83, Fr83], Ada, Exchange

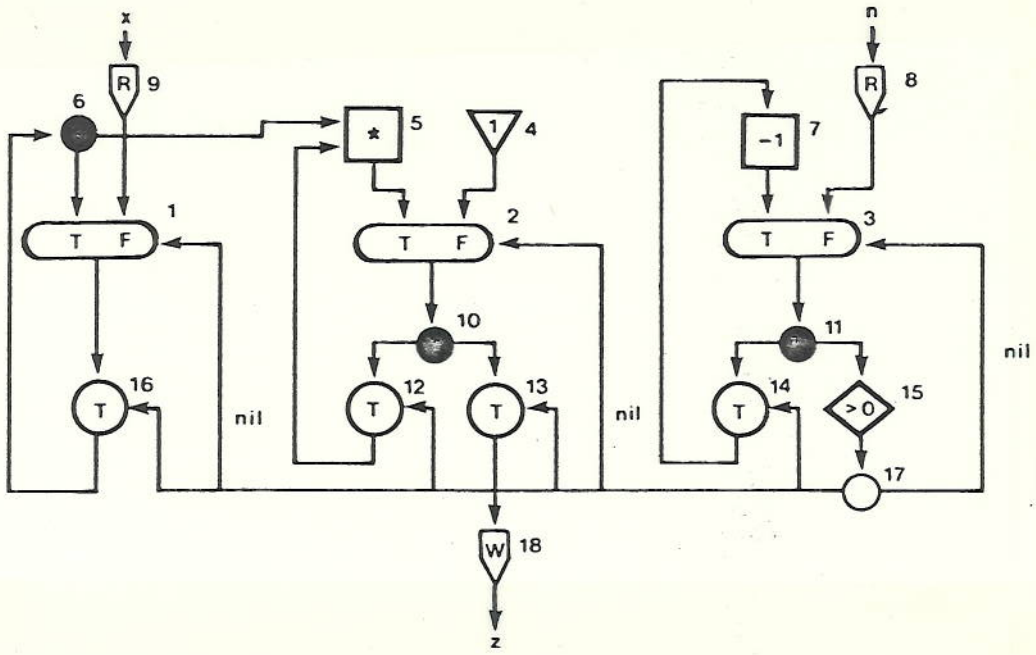

Figure 2. A program for computing $z = x^n$

Functions [Fi77], and Edison [Br81] have all shown similar compactness. The implication is that Scheme is well suited to the operational semantic modeling of languages involving parallel computation. This is not surprising because Scheme has the full power of classical denotational semantics: normal order can be obtained by using thunks and continuations can be accessed directly. However, a formal specification of the semantics of the parallelism supported by Scheme is still under development.

The prototyping mechanism we introduced is grounded largely on Scheme's syntactic extension facility and its basic parallel processing primitives. What is important here is that we transform data into programs instead of introducing an interpreting mechanism especially for the Data Flow model. A system that does not need a run-time interpreter has many positive performance advantages. This technique may not be unique to Scheme, but the naturalness of the prototyping technique is well exemplified through Scheme.

§6 Acknowledgements

We would like to thank Mitchell Wand, Christopher T. Haynes, Robert E. Filman, Eugene Kohlbecker, and Richard Salter for their valuable suggestions during the development of this paper.