DATA FLOW IMPLEMENTATIONS OF A LUCID-LIKE PROGRAMMING LANGUAGE

by

Andrew Lawrence Wendelborn, B.Sc.(Hons.)

A thesis submitted for the degree of

Doctor of Philosophy

in the Department of Computer Science

University of Adelaide

February 28th 1985

Awarded 10-6-85

CONTENTS

LIST OF FIGURES AND TABLES

SUMMARY

The programming language Lucid is a nonprocedural language proposed by Ashcroft and Wadge. In their formal definition, a Lucid program is regarded as a set of equations expressing relationships between infinite sequences of data objects; the solution of the set of equations is defined using least fixed point semantics. The formal definition defines a family of programming languages, with each member determined by a specific data algebra. This thesis defines a particular Lucid-like language, LX, with list structures, strong typing and clause-oriented syntax.

Experiments in the translation to data flow graphs of the language LX are of primary concern in the thesis. An extended data flow model, derived from the work of Dennis, is presented, and an interpreter described. The operations of the model are low level, in order to exploit instruction-level parallelism. Operations are executable when all input operands are present; in other words, it is a data-driven model.

Demand-driven data flow, in which computation can be likened to call-by-need evaluation, is more appropriate to the correct implementation of the formal semantics than is data-driven data flow, which is essentially call-by-value. A technique is developed whereby demand-driven computation is modelled by representing demands as data. An operational semantic model of LX is presented, which describes an abstract computation in terms of the flow of demands initiated by a request for a program result. The semantics of each LX language construct is expresssed as a demand transformation, specifying the demands propagated on receipt of a demand for a particular value. A detailed description of a translator for LX which produces demand-driven graphs is given.

An implementation strategy more directly related to data-driven data flow is also considered. A subset of LX, called LX3, is defined which allows a simple operational interpretation in terms of loops. Techniques for the translation of LX3 to data flow networks are proposed, and their implementation described. The thesis shows that LX3 can be implemented simply and efficiently, and that LX3 is comparable in expressiveness to high level languages developed specifically for data flow. An implementation of LX3 for a conventional, sequential machine is also described.

The demand-driven code generated by the LX translator implements the formal definition, but incurs significant overhead in treating demands as data, and in invoking a separate computation for each value required. The application to LX of known optimization techniques for the latter problem is discussed. However, the thesis emphasizes an alternative approach to performance optimization, whereby the LX and LX3 translation techniques are combined. An extension to LX3 allows some definitions to be written in LX; a translator for the extended language would produce data flow graphs with both demand-driven and data-driven components. The interface between these components utilizes early completion data structures to enable communication between the components. A possible implementation technique is discussed, with an example of its application.

The thesis includes descriptions of Pascal programs which implement the data flow graph interpreter and translators for LX and LX3; these programs have been used to thoroughly test the translation schemes proposed. Specific examples of the operation of each implementation are demonstrated and analyzed.

DECLARATION

This thesis contains no material which has been accepted for the award of any other degree or diploma in any University, and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

If this thesis is accepted for the award of the degree, permission is granted for it to be photo–copied.

A.L. Wendelborn

February 28th 1985

ACKNOWLEDGEMENTS

I am extremely grateful to my supervisor, Dr. Barbara Kidman, for the invaluable discussion, encouragement and criticism she has provided, and for meticulously reading many drafts of this thesis. I also thank Dr. C.D. Marlin for useful discussions and comments, and Prof. C.J. Barter for acting as supervisor while Dr. Kidman was on leave. Assistance from colleagues at the University of Adelaide and at Flinders University is also acknowledged.

I am deeply indebted to Rosemary, for her patience, understanding and moral support during the course of this work, and also for preparing the diagrams used in the thesis. Thanks also to Damian, for tolerating frequent absences from home.

This research was supported in part by a Commonwealth of Australia Postgraduate Research Award.

CHAPTER 1

INTRODUCTION

1.1 Aims of the thesis

Lucid [AshW76, AshW77a, AshW79a, AshW79b] is a functional language with a well defined formal semantics. This thesis reports on research into the design of a programming language, LX, based on Lucid, and on experiments in its implementation. Published descriptions of Lucid give the semantics of the language in detail, but other aspects of language design are given less detailed treatment. Hence, in this thesis, previous descriptions of Lucid are used almost unchanged in defining the semantics of LX. An aim of the thesis is to design a suitable concrete syntax taking into account such factors as clarity and correctness of programming in the language, and ease of compilation. Thus facilities for structuring data and programs, for controlling scope and for type checking, are included.

However, the thesis is concerned primarily with the implementation of LX. The aim of the earliest experiment in the research reported here (and also in [Wen81, Wen82]) was to establish that a Lucid-like programming language could be implemented, with acceptable efficiency, using essentially conventional compiler construction techniques [Wir76], combined with dependency analysis. This experiment led to the development of the implementation structure used in subsequent experiments.

The main aim of later experiments was to explore relationships between LX and data driven data flow [Den74, ArvGP78, GurGK81]; the nodes of a data flow network are low level, side-effect free operations activated by the availability of operand values flowing on the arcs. This was done by constructing two implementations, one emphasizing efficiency of the target code but restricting the language translated, the other, correctness and completness of the translation, in the sense of agreement with the mathematical semantics. For the first, the thesis describes restrictions on LX to give LX3, a subset which can be implemented efficiently, and which is comparable in expressiveness to certain other languages developed specifically for data flow.

An important research goal was the development of an implementation of unrestricted LX with data driven data flow as the target language. A relationship between Lucid and data flow was noted by Ashcroft and Wadge in [AshW77a], in which they use the term "data flow" with respect to networks of the type proposed by Kahn [Kah74]. These networks describe data flow at a higher level (that is, the nodes represent more complex operations) than the networks of principal concern in this thesis. In addition, Ashcroft and Wadge note that not all valid programs can be expressed using such networks, but that an implementation based on the notion of demand driven computation is correct. This suggests that demand driven data flow might be used as the basis of a complete, correct implementation. The work of this thesis differs in that the data flow networks considered are low level and data driven, hence it was necessary to resolve the apparent differences between the data and demand driven views. To achieve this, the thesis presents a new operational semantic definition of LX which explicitly models the flow of demands during a hypothetical computation yielding the same result as specified by the formal definition of Lucid.

The second implementation of LX is a literal application of this operational semantic definition; because it explicitly represents demands and demand flow, it has inherent inefficiencies. However, it provides a precise and readily comprehensible operational model of program execution in a data driven data flow environment; several possible optimizations for improving the performance of the implementation are indicated. Optimization is not the only approach to performance improvement, and the thesis concentrates on an alternative approach. A third, hybrid, implementation, is proposed, in which the first implementation method is used primarily, with the second applied to program components when so specified by the programmer.

1.2 An introduction to LX

This section attempts to illustrate the basic concepts of LX by explaining some example programs. The first program considered is given in Figure 1.1. It demonstrates that a program is an unordered set of definitions; Lucid and LX are equational programming languages. In the figure, the program is shown on the left, with the values of some program identifiers and constants given on the right. The variable i is defined as the history of even numbers. In LX, the variable ***result***, in a program, is regarded as the output of the program. In this case, the output is formed by adding 1 to each value in the history $[\![i]\!]$.

```
prog OddNos;
  int result, i;
  result = i + 1;                 [[1]] = ⟨ 1, 1, 1, ... ⟩
  i = 0 fby i + 2;                [[i]] = ⟨ 0, 2, 4, ... ⟩
eprog                        [[result]] = ⟨ 1, 3, 5, ... ⟩
```

Figure 1.1. Program *OddNos*.

The program illustrates several other points about LX and Lucid. Firstly, variables denote histories, where a history is an infinite sequence of values. The notation $[\![a]\!]$ means "the history denoted by a", and $[\![a]\!]_t$ should be read "the history denoted by a at time t", written thus:

$$[\![a]\!] \quad = \quad \langle\ [\![a]\!]_0,\ [\![a]\!]_1,\ [\![a]\!]_2,\ \ldots\ \rangle.$$

When the meaning is clear, this may be abbreviated to:

$$[\![a]\!] \quad = \quad \langle\ a_0,\ a_1,\ a_2,\ \ldots\ \rangle.$$

Secondly, arithmetic operations apply pointwise to histories, for example:

$$[\![a + b]\!] \quad = \quad \langle\ a_0{+}b_0,\ a_1{+}b_1,\ a_1{+}b_1,\ \ldots\ \rangle.$$

The operator **fby** (pronounced "followed by") is a special operator defined thus:

$$[\![a \ \mathbf{fby}\ b]\!] \quad = \quad \langle\ a_0,\ b_0,\ b_1,\ \ldots\ \rangle.$$

A variable, such as i, defined by an expression involving **fby**, is referred to as an inductive variable.

The next example, the program *Sums* in Figure 1.2, uses a variable n as input; its output at time t, as defined by ***result***, is the sum of the input values n_0, n_1, …, n_t.

The variable n is declared as a global of the main program; in LX, the values in the history denoted by such a variable are to be supplied as input from an external source.

```
prog Sums global int n;              [[n]] = ⟨ 2, 7,  3,  8, ... ⟩
   int result, sum;
   result = sum;                    [[sum]] = ⟨ 2, 9, 12, 20, ... ⟩
   sum = first n fby sum + next n
eprog                            [[result]] = ⟨ 2, 9, 12, 20, ... ⟩
```

Figure 1.2. Program *Sums*.

The monadic operators **first** and **next** are defined thus:

$$\begin{aligned} [\![\textbf{first}\ a]\!] &= \langle\, a_0,\ a_0,\ a_0, \ldots \,\rangle \\ [\![\textbf{next}\ a]\!] &= \langle\, a_1,\ a_2,\ a_3, \ldots \,\rangle. \end{aligned}$$

From the definition of **fby**, it can be seen that, in program *Sums*

$$\begin{aligned} [\![sum]\!]_0 &= [\![\textbf{first}\ n]\!]_0 \\ &= n_0 \end{aligned}$$

and also that

$$\begin{aligned} [\![sum]\!]_{i+1} &= [\![sum + \textbf{next}\ n]\!]_i \\ &= [\![sum]\!]_i + [\![n]\!]_{i+1}. \end{aligned}$$

The definition of *sum* can be visualized as describing an infinite loop in which the variable is initialized to the first input value (n_0), with subsequent values computed by adding the current value of *sum* to the next input value.

The program *NM* in Figure 1.3 defines as its result the square root of the input a. It is convenient to regard the main program variable ***result*** of program *NM* as being defined point by point by the **define** clause *SqRt* invoked with the input, a, as its actual parameter. At time point t, *SqRt* defines (outer) $[\![result]\!]_t$ as follows. The formal parameter r is declared as frozen; this means that, within the clause as invoked, r denotes the constant history $\langle\, a_t,\ a_t,\ a_t, \ldots \,\rangle$ formed by freezing the actual parameter at the outer time, t. The square root is defined by forming x, denoting a history of approximations to the square root, and using the **asa_then_easa** ("as soon as") operator to extract the tenth approximation. The definition of this operator, operating on two variables a and b, is

$$[\![\textbf{asa } b \textbf{ then } a \textbf{ easa}]\!] = \langle\, [\![a]\!]_s,\ [\![a]\!]_s,\ [\![a]\!]_s,\ \ldots \rangle$$

where s is such that

$$[\![b]\!]_i = \text{true},\ i{=}s$$

and

$$[\![b]\!]_i = \text{false},\ i{<}s.$$

From this definition, it can be seen that (inner) ***result*** denotes the constant history $\langle\, x_9, x_9, x_9, \ldots \rangle$. This value then defines outer result at time t, as required. Operationally, the **define** clause can be seen as an inner loop executed 10 times at each step of an outer loop; each iteration of the outer loop accepts a value from input and produces its square root. A definition which uses **fby** models an infinite iteration; the **asa_then_easa** operator models the termination of a loop.

The inner loop (the **define** clause) can be seen as having been invoked at each step of the outer loop, with the histories denoted by variables inherited from the outer loop (a in the example) regarded as constant (frozen) for the duration of the inner loop. It is in this way that LX, like Lucid, models a nested subcomputation.

```
prog NM global real a;                   [[a]] = ⟨ 4, 16, 9, ...  ⟩
   int result, SqRt;
   define SqRt( real r ) freezing r;
      real x, result; int Count;
      result = asa Count eq 10 then x easa;
      x = 1 fby (x + first r / x) / 2;
      Count = 1 fby Count + 1
   edefine;
   result = SqRt( a );
eprog                                [[result]] = ⟨ 2,  4, 3, ...  ⟩
```

Figure 1.3. Program *NM*.

1.3 Structure of the thesis

The next section of this chapter surveys the historical development of Lucid, and examines its relationship to LX. A description of data flow models relevant to the thesis follows. Important issues in implementing LX are then discussed.

Chapter 2 formally defines the syntax of the language LX. The semantics of LX are explained informally by comparison with the semantic definition of Structured Lucid [AshW79b], and defined operationally using an information structure model which describes the flow of demands in a computation. The loop based subset, LX3, is then defined by giving restrictions on LX programs, and showing how such programs can be interpreted in terms of loops. Chapter 3 describes the data flow model used in the thesis.

In Chapter 4, the implementation of LX3 is described. The importance of dependency analysis in determining the loop structure of a program is demonstrated, and appropriate data flow schemes are described, corresponding to LX3 constructs. A new loop scheme is presented, of general applicability in data flow languages, which permits reference to both current and next values of an inductive variable.

The demand driven implementation is described in Chapter 5, by describing how appropriate data flow schemes can be developed from the operational semantic model. Chapter 6 shows how the two implementations can be combined so as to allow the schemes of the second implementation to be used, with certain restrictions, in LX3 programs.

Chapter 7 discusses various aspects of the thesis, in particular, relationships between the implementation of LX and other implementations, of Lucid-like languages, which use data flow concepts. It also outlines areas of possible future work, in particular, suggestions for improved input/output facilities in both LX and data flow.

1.4 The Development of Lucid

Lucid originated as a formal system, proposed by E.A. Ashcroft of the University of Waterloo, and W.W. Wadge, of the University of Warwick. Three distinct stages in the development of Lucid can be distinguished. The original proposal, termed Basic Lucid [AshW76, AshW77a], was published in 1976. A short time later, an extended language, which will be referred to as Clause Lucid, was proposed [AshW78]; it permits functions to be defined, thereby giving control over the scope of identifiers. This proposal was modified somewhat with the advent of Structured Lucid [AshW79a, AshW79b] in 1979, in which Ashcroft and Wadge describe Structured Lucid in two distinct parts. Firstly, USWIM, a logical programming language based on Landin's ISWIM, is defined, allowing the expression of structured, recursive definitions. Secondly, this language is combined with Basic Lucid to give Structured Lucid, a language which includes iteration, recursion and function definition and is defined by a simple denotational semantics. The latter definition is a significant advance over Basic and Clause Lucid in that it constitutes a clearer and more unified mathematical description of the language.

Each of these stages will now be described in more detail.

1.4.1 Basic Lucid

In [AshW76], Ashcroft and Wadge state that "Lucid is both a language in which programs can be written, and a formal system for proving properties of such programs". A Basic Lucid program is simply an unordered set of assertions, from which other assertions may be derived. Programs are denotational and referentially transparent.

In [AshW76], a program is defined as a set of equations, with the right hand side of each essentially an expression comprising variables combined using logical, integer and special Lucid operators. The latter include **first**, **next**, **as soon as**, **latest** and **followed by**. A variable x must be defined exactly once (except the variable ***input***, which is assumed to be defined outside the program), in one of the following ways:

$$\begin{array}{lrcl}
(1) & x & = & E \\
(2) & \textbf{latest}\ x & = & E \\
(3) & \textbf{first}\ x & = & E_1 \\
 & \textbf{next}\ x & = & E_2.
\end{array}$$

In the above, E, E_1 and E_2 are expressions. In Basic Lucid, the operator **latest** is used to express nesting of subcomputations and freezing. Form (3) is the definition of an inductive variable, using two equations; in LX (and later versions of Lucid), it would be written

$$x \;=\; E_1 \textbf{ fby } E_2.$$

Variables and expressions denote infinite sequences of data objects, and equations are assertions about the histories of variables. All operators in the language are interpreted as operations on infinite sequences. Definition (3) models an infinite iteration, in that it specifies an initial value in the history of x, and an expression giving the next value in the history, often in terms of the current value. The **as soon as** operator makes it possible to extract a value from such an infinite iteration.

In Basic Lucid, the history of a variable used in a nested loop is an infinite sequence, each member of which is also an infinite sequence, and so on, for further levels of nesting. Hence, nested iteration is allowed for by considering each history to be function of an infinite sequence of time parameters. The first member of the sequence of time parameters represents the iteration number of the most deeply nested loop, the second member the enclosing loop, and so on; it is clear that only a finite number (equal to the level of nesting) of these parameters is relevant, the rest are added for "convenience."

Under this interpretation of nested iteration, the operator **latest** is defined as

$$\textbf{latest } i \text{ at time } t_0 t_1 t_2 \ldots \;=\; i \text{ at time } t_1 t_2 t_3 \ldots$$

giving access to an additional time parameter, or level of nesting; use of **latest** i on the right hand side of an equation gives access to i in the immediately enclosing iteration, effectively making i global to the loop. Use on the left hand side (form (2) above) has the effect of passing the value out of the loop. To avoid explicit use of **latest**, Basic Lucid has the program structuring construct

```
begin
    ...set of equations...
end
```

which has the effect of applying **latest** to every use of a global variable, (that is, a variable which is not local) between the **begin** ... **end** brackets. A use of a global on the right hand side of an equation always yields the value of its operand in the immediately enclosing iteration; this has the effect of freezing the operand within the inner iteration.

The semantic definition given in [AshW76] formalizes these ideas in terms of mathematical objects termed computation structures, which are power structures based on first-order logical structures. It is also shown that every program has a minimal solution, given a suitable approximation ordering on the underlying data values.

Ashcroft and Wadge [AshW76, AshW77a] show that the formal semantics leads to several inference rules which can be used to construct proofs of programs.

1.4.2 Clause Lucid

In [AshW78], Ashcroft and Wadge describe extensions to Basic Lucid which allow functions to be defined and the scope of variables to be restricted. This is achieved by the introduction of four clauses, the **produce**, **function**, **compute** and **mapping** clauses. The **compute** clause is equivalent in effect to the **begin ... end** construct of Basic Lucid. **Produce** and **function** clauses, in programs, take the form

```
produce VARIABLE using VARIABLE_LIST
    ...set of assertions...
end

function VARIABLE (VARIABLE_LIST) using VARIABLE_LIST
    ...set of assertions...
end.
```

In both cases, the set of assertions must include a definition of the local variable ***output***; the variable at the head of the clause is termed the *subject* of the clause, and the clause constitutes a definition of its subject. The variables in the using list are the globals of the clause, and the variables listed within parentheses are the formal parameters. The **compute** and **mapping** clauses are syntactically similar to the **produce** and **function** clauses, respectively, with the set of assertions including a definition of the local variable ***result***, instead of ***output***.

In Basic Lucid a program is regarded as a set of assertions; in Clause Lucid, this approach is extended so that clauses are defined as compound assertions about their local and global identifiers. Within a clause, ***output*** refers to the subject of the clause; for **compute** and **mapping** clauses, ***result*** means **latest** ***output***, and **latest** is automatically applied to all globals and formals of the clause.

In Basic Lucid, a variable denotes a history, which, as noted above, is an infinitely dimensioned infinite sequence of data values. In the extended semantics of Clause Lucid, the subject of a **function** with n formal parameters denotes an n-ary semantic function from histories to histories; the complete formal parameter history is used in determining the meaning of the function. The subject of a **mapping** with n formal parameters denotes a stream of n-ary data functions; that is, the meaning of the mapping is defined point by point, using parameter values at that point only, hence **mapping** application is the pointwise extension of conventional function application. It follows that, in the case of a **produce** or **function** clause, a global denotes the same history inside as outside the clause, whereas, within a **mapping** or **compute** clause, it denotes the **latest** value of the global; in other words, globals and parameters can be regarded as frozen inside **mapping** and **compute** clauses.

In [AshW78], additional rules of inference, derived from the formal semantics, are introduced for reasoning with clauses. Rules for program transformation are also presented; under certain conditions, it is possible to add assertions to a clause, move assertions into, or out of, a clause, or rename the local variables of a clause. Ashcroft and Wadge show that the language does not use call by value in passing parameters, and that it uses static rather than dynamic binding.

Ashcroft and Wadge also give possible operational interpretations of each clause. In the case of a **produce** clause, globals have the same meaning inside as outside a clause; definitions are at the same "level of iteration" both inside and outside the clause. This suggests that a **produce** clause can be seen as a coroutine-like block of code "which is repeatedly executed but with persistent internal memory in the form of inductively defined local variables" [AshW78]. It can also be seen as "an ongoing process which continuously produces values of its subject" [AshW78]. **Function** clauses can then be seen as templates for such processes, which each textual occurrence of a function call giving rise to a new instance of the template. In the case of recursive calls, this requires dynamic creation of new processes.

The **compute** clause, because it freezes its globals, can be seen as a nested loop. The **mapping** clause defines a pointwise function of its arguments and globals; a use of a **mapping** clause corresponds to a conventional function call.

1.4.3 Structured Lucid

Structured Lucid [AshW79b] was developed by using a different approach to the definition of Lucid. The approach taken is to first define a denotational language, USWIM, which permits recursive definitions and program structuring [AshW79a]. USWIM is similar to many other assertional languages, and gives a well defined basis for defining Structured Lucid by adding the Basic Lucid concept of iteration. The result is a language which expresses essentially the same semantic ideas as Clause Lucid, but with a much simpler mathematical presentation of the semantics. Its development is also more clearly related to earlier ideas in denotational languages.

The language USWIM is a variant of Landin's ISWIM [Lan66a], and, like ISWIM, actually defines a family of programming languages; USWIM defines "the ways of expressing things in terms of other things" [AshW79a], with the set of primitive things being specified as an algebra, which determines a particular member of the USWIM family.

According to the abstract syntax given in [AshW79a], a USWIM program is a term, where a term is either a variable, an expression or a phrase. The **valof** phrase provides a means of structuring programs, as in the following example

```
valof
  result = f(5)
  f(x) = x + valof
                 g(z) = if z< 1 then 1 else z*g(z - 1)
                 c = x * x
                 result = g(x) + c
               end
end.
```

The **valof** phrase represents the major difference between USWIM and ISWIM; it corresponds to the **whererec** phrase of the latter. The example also illustrates the differences between a phrase and a clause. A phrase is essentially expression-oriented, whereas a clause is definition oriented. A clause is used as a compound definition, defining an n-ary variable which can then be used in an expression. The definition of f shows the manner of definition of an n-ary variable in a phrase-oriented language. It also shows the use of an anonymous phrase in the expression which makes up the right hand side of the definition of f.

The formal semantics of USWIM defines how the value of a term can be determined from an algebra and an environment; an environment is a function which assigns to each n-ary (where n is the number of formal parameters of the definition of the variable) variable an n-ary semantic function over the universe of the underlying algebra. For example, a 0-ary variable is assigned a 0-ary history function which maps the variable to a history, where a history is an infinite sequence of values from the underlying algebra.

The semantics can be used to justify various syntactic manipulation rules. Rules are presented for substitution, the importation of variables, renaming of local variables, function calling, and others. By embedding it in a suitable first order logical system, Ashcroft and Wadge [AshW79a] extend the language to permit the manipulation of assertions.

USWIM forms constructs from equations; Basic Lucid allows iteration in a mathematical way, using simple equations but a modified data algebra. The essential idea behind Structured Lucid [AshW79b] is that the two can be combined, in a denotational framework, to give a language which combines program structuring facilties with the ability to write both recursive and iterative definitions.

Ashcroft and Wadge [AshW79b] discuss two ways in which the combination can be made. The first is to form a member of the USWIM family with an algebra similar to that of Basic Lucid, with variables denoting histories; the language which results is termed ULU. In this language, globals have the same meaning inside a phrase as outside. In fact, an ULU definition which uses a **valof** phrase as its right hand side is analogous to either the **produce** or **function** clause of Clause Lucid, depending on whether or not the left hand side has formal parameters. The operational interpretations of ULU programs are also similar.

A second way in which the ideas of USWIM and Basic Lucid can be combined is to use a pointwise extension of USWIM (with a standard data algebra). The language formed in this way is called LUSWIM. In LUSWIM, a single outer environment does not determine a single inner environment (as it does in ULU), but a sequence of environments, one for each outer time step. Each inner environment is obtained by freezing the outer environment at a particular time. It is required that each variable used in a LUSWIM program denote an elementary history function, which is a function with the property that its value at any time depends only on the value of its arguments at that time, and perhaps the time itself; it is possible to freeze such a function at a particular point in time. Certain restrictions are imposed on the syntax of definitions to ensure that all LUSWIM variables are elementary (an elementary variable denotes an elementary history function). This ensures that all variables can be frozen in the manner required. LUSWIM phrases are similar in effect to the **compute** and **mapping** clauses of Clause Lucid, and have similar operational interpretations.

Structured Lucid is the language formed by combining ULU and LUSWIM. In this language, there are two classes of variable, elementary and non-elementary. The elementary variables must be defined according to the restrictions applicable to LUSWIM, and are subject to freezing inside phrases. All other variables are non-elementary; phrases which use only such variables can be understood in the same way as ULU phrases.

The formal semantics combines the two languages by defining the result of a **valof** phrase pointwise, with the environment frozen at each point, as defined below. Freezing of globals in a phrase is defined formally by saying that the meaning of a phrase in a Lucid environment E is a history a, where a is defined point by point, thus: a_s is defined as the meaning of ***result*** at time s, where ***result*** is determined in a frozen environment E' which incorporates the outer environment E frozen at time s. The definition of freezing an environment captures the distinction between elementary and non-elementary variables by specifying that the latter are unaffected by freezing (that is, the frozen environment assigns to them the same meaning as unfrozen), whereas the frozen environment assigns to an elementary variable the meaning $\langle B_s, B_s, B_s, \ldots \rangle$, where B is its meaning in the unfrozen environment.

This represents a significant simplification in the presentation of the formal semantics compared with that of Clause Lucid; the use of nested environments replaces that of sequences of time parameters, thereby simplifying the notion of history. Consequently, the operator **latest**, and the attendant distinction between ***output*** and ***result*** (**latest** ***output***), are no longer needed.

Structured Lucid permits both frozen and unfrozen globals to be present in one phrase (Clause Lucid allowed one or the other, but not both). Such a phrase can be considered operationally as basically an ordinary LUSWIM phrase in which the non-elementary variables must be thought of as being restarted at the beginning of every subcomputation.

1.4.4 Relationship of LX to Lucid

The language LX described in this thesis has the same semantics as Structured Lucid, but is significantly different in syntax. Firstly, the grammar has been adapted to make it LL(1). LX, while clause based, has only a single kind of clause, the form of which is simpler, and, it is hoped, clearer than equivalent Structured Lucid phrases. The language requires explicit specification of variables frozen within a clause, and of those global to a clause (with a distinct form of specification for those global to the main program). Strong typing is used—the type of every variable must be specified, all values in the history denoted by a variable are of the same type, and every expression has a data type which can be determined statically. The structure type provided in

LX is a linear list structure with elements of the same particular basic type.

1.5 Data flow models of computation

A broad interpretation of the term "data flow" is that it describes a system in which the actions which take place are determined primarily by the flow of data within the system; the system is driven by the passage of data. Conway's original proposal for implicitly sequenced coroutines [Con63] can be seen as defining a language facility for the construction of data driven programs. Some more recent language proposals for distributed systems specify a system of processes which communicate by passing messages [Hoa78]; although each process is sequential (executed under control flow), the resumption and suspension of processes is determined by the flow of messages (data), and is thus data driven at the process level.

The data flow systems considered in this thesis are those which are data driven at the operation level. The fundamental principle of data flow computation at this level is that an operation is enabled for execution whenever its operands are available. In a system in which this is the only rule for determining executable operations, program execution is entirely data driven; most systems, however, incorporate some modifications to this rule. In addition, operations are not permitted to have side effects, hence all enabled operations can be executed concurrently. Several proposals based on the data flow model of computation have been put forward for machine architectures capable of exploiting this parallelism.

Most early work on data flow models and architectures was carried out at MIT [Rod69, Den74, DenFL74, Mis77]. Other early proposals were put forward by Karp and Miller [KarM66], and Adams [Ada71], and the LAU system [ComHS80, Syr82] is based on data flow principles. Important data flow projects were also initiated at the University of California at Irvine [ArvGP78], Manchester University [GurWG80, GurGK81], Iowa State University [OldTRZ77, AllO80], the Univerity of Newcastle upon Tyne [TreHR82, TreBH82] and Westfield College [HanG81]. More recently, Japanese researchers have been active in the data flow area [Ama82, ACM82], working as part of the so-called "Fifth Generation" project [Tan82, ACM82]. Surveys of work in data flow and related areas can be found in [TreBH82, ArvA82, Den80].

Some characteristics of the data flow computational models relevant to this thesis will now be discussed. An important model, which has had considerable influence on subsequent proposals, is that of Dennis [Den74]; the model, often termed DDF (Dennis data flow), is now described in some detail, and used as the basis of comparison with other models.

A data flow program is a directed graph, the nodes of which represent operations, and the arcs, channels on which value-carrying tokens flow. Dennis distinguishes between control arcs, which carry Boolean values, and data arcs, which transmit values of type integer, real or string.

Dennis classifies a node, according to the values on which it operates, as a fork, an operator, a decider, a Boolean node or a control node. The terminology used here is slightly different from that of Dennis, who classifies nodes as links and actors; a "link" is referred to here as a fork, while "actors" encompass the remaining node categories. A fork permits an incoming value to be copied onto one or more output arcs (in Dennis' model, a fork, or link, can be seen as a cell storing the value which occupies the arc; here, a fork is regarded as a replicator of values). An operator node acts on one or more data values; typical operators are plus, minus and other arithmetic operations. A decider (predicate) produces a control value from one or more data values. A Boolean,or control, node operates on control values; examples are the operations AND, OR and NOT. The control nodes, MERGE, TGATE and FGATE, use control values to regulate the flow of data values.

In general, a node is enabled for execution (firing) when a value is present on each input arc, and, in addition, each output arc is empty; the input values are absorbed, the appropriate computation performed, and the result transmitted on the output arc.

The control nodes behave somewhat differently. The TGATE instruction takes control and data inputs, and becomes enabled when both are available, as usual; on execution, the data value is transmitted (unchanged) if the control input is true, otherwise it is absorbed and no output is produced (the FGATE instruction behaves similarly for a false control input). The MERGE instruction fires when its control input and the corresponding data input are present (the data inputs are labelled T and F); the data value is transmitted, the other input line being unaffected. An initial value may also be specified (that is, a value placed on an arc before initiation of execution of the data flow program) on the control input of a MERGE instruction. Control nodes are used in constructing conditional and iterative schemes.

Dennis also proposes facilities for tree structures in the language, manipulated by SELECT and APPEND operations. Conceptually, structure values flow on arcs of the data flow graph in the same way as other values, with all operations creating new values rather than updating existing ones (to ensure the absence of side effects). Dennis extends the model to include a heap, with structure values represented as heap nodes, identified by pointers into the heap. In this representation, pointers, rather then actual structures, are transmitted on the arcs of the graph, and structure operations cause changes to be made to the heap; this allows structures to be shared.

A data flow procedure is a data flow graph which accepts, as input, a structure containing parameter values, and produces a value from those parameters. The APPLY node permits such procedures to be activated. It accepts two parameters, the data flow graph of the procedure (regarded as a node, containing a representation of the code of the procedure, on the heap), and the parameter structure, and produces, as its output, the result of the procedure. The procedure is activated, each time APPLY fires, by putting a token on its input arc. It is clearly necessary to distinguish tokens which arise from different firings of APPLY; the notion of token colouring is used to achieve this. Every token is tagged with a colour, which identifies a particular activation. The token which initiates a new activation is given a new colour; the colour of the result of the procedure is restored on termination. The general firing rule is modified so that a node is enabled if values of the same colour are available at each input; the output arc must contain no value of that colour, but it can contain values of other colours.

Researchers at the University of California at Irvine extended Dennis' model (DDF) to permit greater asynchcrony in the execution of operations [ArvG78, ArvGP78]. DDF specifies that all output arcs of a node must be empty for the node to be enabled for firing. This rule prevents conflicts between the tokens at the inputs of a node, at the expense of restricting the number of enabled nodes; the rule enforces a queuing discipline on the transmission of tokens along an arc, restricting the queue size to one for practical reasons. Arvind and Gostelow [ArvG78] describe a data flow language very similar to DDF, and present a queuing interpreter (QI) in which the arcs are unrestricted FIFO queues. Under QI, the operation

$$a_i \mathbin{++} b_i$$

must be preceded by

$$a_{i-1} \mathbin{++} b_{i-1}$$

where the subscripts index the queues on the input arcs of the node ++; the firing of the node is not restricted by the state of its output arcs.

Arvind and Gostelow also describe an unravelling interpreter (UI) under which the firings of a node need not take place in queue order. The i^{th} output of a node can be produced as soon as the i^{th} set of inputs is available, even if the $i - 1^{th}$ set is not complete. This permits greater asynchrony of operation, and, in cases where computation of the $i - 1^{th}$ set does not terminate, produces results that QI fails to produce.

Under UI, each distinct execution of an operator is regarded as an independent activity [ArvGP78]. Each activity has a unique name, comprising the procedure application context, the procedure name, the node within the procedure, and the iteration number. The context acts as a stack of return addresses, with an address pushed on procedure invocation, and popped on exit. All the activities associated with a procedure invocation constitute a procedure domain, and can proceed independently. A loop scheme is similar to a procedure invoked from just one place; the iteration number is used to distinguish activities within a loop domain. The context and iteration number fields are further examples of token colouring. The model can be classified as a dynamic tagged model [WatG82], in that each token is tagged with a colour; it is dynamic in the sense that colouring enables a procedure to be used re-entrantly, as distinct from static data flow [DenGT84], in which only one instance of an instruction may be active at a time.

The Id system [ArvGP78] includes a representation of the stream data structure as an ordered sequence of tokens; streams permit the expression of history sensitive computations. Facilities for the expression of nondeterminism and resource managers, and for programmer defined data types, are also proposed. Streams will be discussed in more detail in §1.6.

The data flow model used by a group at the University of Manchester [GurWG80] is also a dynamic tagged model, but was developed independently of the Irvine model. In the latter model, an activity name records a stack of procedure and loop contexts; the Manchester model uses a fixed length label for each token, in which procedure invocations are identified by a unique activation name, and iterations by an iteration number field. To achieve this, it is necessary to have a special instruction to generate activation names (which cannot be defined as a simple function of a previous name, because of the possibility of parallel invocations), and to express nested iterations as procedures (as only one iteration number is allowed in the label). In addition, nodes are restricted to at most two inputs and/or outputs. All of these characteristics of the model are influenced by practical consideration for the efficiency of the associated machine architecture. Structures are regarded, not as entities flowing along an arc or residing in a heap, but as individual tokens related by having identical labels, except for the value of an additional, index field. This scheme is used to implement an array structure [GurWG80].

For a two operand node to be enabled, operands with matching labels must be present. The Manchester model has been extended by associating varying actions with the process of matching potential operands (see below). This approach has been used to optimize structure storage and access, and, with the addition of a nondeterministic operator, to implement resource managers [CatG80]. A prototype machine has been built at Manchester; an evaluation of its performance appears in [GurW83].

The data flow models described above are the most significant from the point of view of this thesis. Several other models have been proposed over the years [Gel76, Syr82, Dav79, Dav78, ReqM83, Gaj81, Sri81, Fau82, HanG81, FarGT79, TreBH82, TreH81, TreHR82]; some of these are now described briefly.

The LAU project at Toulouse [Gel76, Syr82] began, in 1973, as an investigation of single assignment languages [TesE68, Cha71, Kli72], and led to the design of a language based on the principle of single assignment and a supporting data flow machine architecture. The single assignment rule states that any variable in a program may be assigned to at most once during program execution; the LAU group uses the term "object" instead of "variable" to emphasize that it is, in fact, constant once assigned. The high level language includes constructs for assignment, conditional, iteration, procedure definition and use, and parallel array computation. The semantics of the language is expressed using the Data Production Set (DPS) concept, where a DPS is a triple specifying a set of input objects, a set of statements which manipulate the inputs, and a set of output objects; data flow principles determine the eligibility for execution of a DPS. It is, of course, possible for many DPS's to be eligible for execution in parallel. The machine architecture implements the DPS concept, and instructions corresponding closely to the constructs of the high level language. A machine has been built, and numerous programs tested.

The model developed by Davis [Dav78] is similar in principle to DDF, with nodes and arcs used to represent a data flow computation graphically. Unlike DDF, no distinction is made between data and control tokens, and the arcs represent data paths which are queues of finite length. The nodes are similar in function to those of DDF, but more general; a network usually contains less nodes than a functionally equivalent DDF network. Davis also introduces a nondeterministic arbiter node for use in resource control. The most significant differences occur in the supporting machine architecture, which is based on the principle of recursive hierarchy [Dav79]. A machine has been built, and is described briefly in [TreBH82].

In a project at the University of Newcastle upon Tyne, the data flow model has been combined with a generalized control flow model (GCF) [TreHR82, TreBH82]. The GCF model uses instructions and memory addresses in much the same way as in the traditional von Neumann model, but uses control tokens to establish multiple threads of control. An instruction is enabled for execution when all its input control tokens are present; enabled instructions can execute concurrently. The execution of an instruction involves absorbing the input tokens, executing the specified operation (which will usually cause the memory to be updated), and generating further control tokens. The program counter of a sequential machine can be seen as encoding a single thread of control in the GCF model.

In the model which combines data flow and GCF, the inputs to an instruction are data and control tokens, as well as embedded items. Data tokens and embedded items can be either values or memory addresses. In executing an instruction, sets of values and addresses are formed from the available inputs; the addresses are then dereferenced to provide, with the other input values, the set of values upon which the instruction operates. The outputs released are of three types: data and control tokens, which specify a destination instruction and a value (null for a control token), and data to be stored in memory, specified as a memory address and a value.

The Piecewise Data Flow project at Lawrence Livermore Laboratories [ReqM83] represents an attempt to combine array processor and data flow principles to give a supercomputer architecture implementable in current technology. The program organization suggested is based on techniques currently used in optimizing compilers, in that a program is divided into basic blocks (of up to 255 instructions) such that no branching into or out of the block occurs. A program is then described at two main levels, the first giving relationships between blocks, and the second those between the instructions of a block. The potential successors of a block are used to decide which block to execute next, and whether or not block execution can be overlapped. Execution of the instructions within a block proceeds according to data dependencies, as in data flow. The machine architecture which supports this program organization uses an array processor, a small number of scalar processors with simple interconnections, and standard input/output techniques.

A proposal by Gajski et al [Gaj81] uses data flow principles, but at a higher level. A program is made up of compound nodes related by data and resource (in particular, memory) dependencies. A compound node represents a function, of higher level than the usual data flow operations, for which good speedup can be had using existing techniques, for example, array operations and linear recurrences.

The Westfield data flow model [HanG81, FreG83] makes use of acyclic graphs, with the nodes representing operators defined at the bit level. The operators of the model are low level, similarly to the DDF model. Functions can be defined, and when used are regarded as being expanded in-line, hence, each node of the dynamic graph fires at most once. A function name is viewed as a value which can be transmitted on an arc. The BIND operator permits a value to be bound to a parameter, constructing a closure; the EVAL operator is used to evaluate a function when all actual parameters have been bound. Higher order functions are supported by these facilities. Structures are represented by functions. The Westfield group have built a prototype uni-processor implementing the model, and developed a corresponding high level language, CAJOLE.

Research into data flow models has also been undertaken at Warwick University [Fau82]. Faustini has investigated the properties of so-called pipeline data flow with respect to the Kahn principle, which states that the operational behaviour of a pure data flow net can be described by the least fixed point solution to the set of equations associated with the net. In Faustini's model, nodes behave in a much more general way than those of the models considered above: there are no restrictions on the possible internal states of a node, state transitions can occur without the arrival of inputs, and transitions may be nondeterministic. Faustini establishes that that nets constructed from such nodes using the operations of juxtaposition and iteration possess the Kahn property.

1.6 Data flow languages

In order to exploit implicit parallelism to the utmost, data flow computational models are based on the following principles:

- the only sequencing constraints are data dependencies, and
- operations are free of side effects.

An important consideration in designing a high level language for data flow is that it reflects these characteristics of the model.

Conventional languages do not have the desired properties. Programs usually need extensive analysis to determine the data dependencies upon which the data flow program is based. Such analysis is made difficult by, among other things, the presence of side effects. Conventional languages are based on a computational model which uses side effects resulting from assignment to memory locations as its fundamental method of communication between statements; because one memory location may be referenced by several statements, data dependencies exist between any statement which updates that location and all statements which reference the location. The problem is compounded by the presence of aliasing, whereby a reference to a location can be passed into a procedure, for example, making it possible for the same location to be accessed via two or more different names. Arbitrary branching in programs adds to the difficulty of determining data dependencies between statements.

Many language facilities have been suggested to improve the undesirable aspects of conventional languages. Examples are the use of structured control constructs in simplifying the structure of programs, and modularity in aiding comprehension by making many data dependencies more explicit. In fact, by imposing restrictions on the way in which programs are written, conventional languages can be made compatible with the data flow model [AllO79, AllO80, Vee81]. However, many designers of languages for data flow have concluded that the difficulties of this approach, combined with a desire for a language in which data flow idioms can be expressed naturally, warrant the development of new languages for data flow.

In looking for a higher level representation of a data flow graph, it seems natural to give a name to an arc of a data flow graph, and describe the values which flow on that arc. Such a description can be seen as assignment to a variable name, but does not make sense if more than one assignment is allowed. Consequently, the single assignment concept [TesE68, Cha71], where a variable is defined at most once in the course of execution of a program, came to be seen as a concept which applied naturally to data flow languages. Programs written in this way express data dependencies clearly.

Data flow languages also require absence of side effects. Applicative languages [Bac78, BurMS81, Bur75, Dar82, Lan66b, Tur81], in which computation proceeds by the application of operations to values, have this property. It is common in applicative languages for information to be passed to a function entirely through its parameter list; this property (locality of effect) makes the detection of data dependencies very straightforward.

Thus, single assignment and applicative languages have properties compatible with data flow principles, and data flow languages therefore have many characteristics inherited from such languages. The data flow computational model has natural representations for conditional, iterative and procedure constructs; many data flow languages [ArvGP78, GurGK81, AckD79] have been influenced by conventional language syntax in this area. A conditional scheme, for example, will use gates, controlled by the result of the condition, to transmit values to one branch and absorb those directed to the other; values must still arrive at both branches, however, and conditional expressions in most data flow languages consequently require that both branches be specified [GurGK81].

A common data flow view of iteration is that of a cyclic graph with a set of inputs, upon which initial values arrive to commence the first iteration. The loop test is used to decide that either a new set of values is directed to the inputs for a new iteration, or the values are used to compute a result for the loop. However, iterations of a loop need not execute successively, because, as mentioned above, some models (for example, [ArvGP78, GurWG80]) permit unravelling of loops during execution, and several iterations of a loop may proceed concurrently. Loop constructs in data flow languages, although syntactically similar to corresponding constructs in imperative languages, invariably use this semantics. Loops can also be seen as tail recursion, and, in some languages (e.g [Wen75]), must be expressed as such. However, iteration offers advantages with respect to efficiency; both the Irvine and the Manchester architectures exploit the characteristics of cyclic schemes at the hardware level. Iteration over structures is discussed below.

Data flow procedures (functions) are, in a sense, an abstraction of a data flow graph as a single node. In the Irvine model, a procedure can accept several input parameters and produce several outputs. In many proposals, procedures are unlike primitive nodes in that they can commence execution before the arrival of all actual parameters; the input interface of the Manchester system has this property. From a practical viewpoint, this allows greater concurrency.

1.6.1 Data structures

In most data flow languages, structures are viewed applicatively—a structured value is a distinct unit, manipulated only by operations applicable to structures; a structure is changed by applying such an operation to produce a new structured value, not an updated version of the original structure. Structure operations thus conform to the data flow principle that operations must be free of side effects. The adverse effect of destructive update in analysis of data dependencies between statements which manipulate structures has been pointed out by Ackerman [Ack82].

The languages VAL [AckD79] and Lapse [GurWG80, GurGK81] provide array and record structures, similar to structuring facilities provided in many conventional languages. Restrictions on the syntax of "assignment" contructs are imposed, reflecting the applicative semantics adopted. For example, in Lapse, assignment must be to the whole array, and at most once, in accordance with the single assignment rule; the language provides a **forall...use...** construct for this purpose, permitting the use of an integer variable which takes all values within the bounds of the array. VAL provides a variety of operations for the creation and manipulation of dynamic arrays, as well as a **forall** construct, for array construction or the specification of an aggregate operation (such as summation) applied in parallel to all elements of the array.

The structures provided in the language Id [ArvGP78] can be described as generalized lists. A structure value is either empty, or a set of ⟨ ***selector*** : ***value*** ⟩ pairs, where ***selector*** is an integer or string value, and ***value*** is any Id value (possibly a structure). Two operators, **Select** and **Append**, are applicable to structures. A one dimensional array can be modelled as a structure with consecutive, integer selectors and non-structured values, and a Lisp list as a structure in which each element has two selectors, for example, the strings "car" and "cdr", and associated values. The language CAJOLE [HanG81] makes no explicit provision for data structures; they are defined as functions, with some syntactic sugar provided for common structures, for example, vectors.

Implementing a structure as a unit can impair the efficiency of operations on that structure. For example, given an iterative algorithm in which certain elements of an array are changed from one iteration to the next, it may be theoretically possible to perform parts of different iterations in parallel; if the array is considered a single value, a given iteration cannot commence until the array value from the previous iteration has been produced, and a source of increased parallelism is lost. Consequently, various proposals have been put forward to permit easier access to the individual components of a structure, while retaining applicative semantics for structure operations.

In the Manchester data flow model, for example, an array is represented as individual elements flowing on an arc, with the index field of a token label representing an array subscript. This permits a straightforward representation of operations such as the addition of corresponding elements in two arrays [GurWG80], but leads to cumbersome schemes, with considerable repetition of code, when scattered elements from one array are used in constructing another [GurGK81]. Storage matching functions [WatG82] can be used to solve this problem, with an array effectively stored at an input of a node, but used in a manner which preserves the applicative semantics of array operations in Lapse.

Streams [Lan65] are structures with the property that the components of the stream are created and used in order, and do not need to exist simultaneously; such structures are clearly useful in communication between pipelined processes. In the context of data flow systems, the use of a stream structure permits access to elements of the structure before the entire structure is created. A stream can be regarded as a structure with two components, ***first***, a value, and ***rest***, a stream, with corresponding selector operations; selection of the ***first*** component can be made without regard for the status of the ***rest*** of the stream. Stated another way, the stream construction function is non-strict in either of its arguments [ArvT81].

Weng [Wen75] first proposed the use of streams in a recursive, acyclic data flow scheme. He represents a stream as a sequence of tokens, terminated by an ***est*** token, and defines several operators for stream manipulation. Most such stream operations have two states, to distinguish the ***first*** and ***rest*** components of a stream. The Id language also supports streams, extending Weng's proposal to allow the use of streams in cyclic schemes, and introducing the **for each** construct to permit processing of each element of a stream; examples are given [ArvGP78] of the use of streams to write history sensitive functions in Id.

In data flow models which use arcs of finite capacity, the "token sequence" representation does not implement streams correctly, in that some programs deadlock unnecessarily [Wen79]. Although Id streams do not have this problem (arcs are conceptually infinite), a "stream of stream" structure cannot be used.

A stream is often used as a buffer between two parallel processes; viewing the stream as a whole, it is apparent that only a few elements are in existence at any one time. The stream can be regarded as a partially completed structure; Weng [Wen79] has proposed an alternative stream representation based on such an idea. The stream **cons** operation creates a binary tree, the left branch of which contains the value representing the **first** component of the stream. The right branch (**rest**) contains a hole, representing a stream which will be filled in as data becomes available. An attempt by a stream consumer to access a hole causes the read request to be held awaiting the arrival of data at the hole; a write-hole operation is provided, which writes a value into a hole, and satisfies any pending read requests. In the DDF model, streams can thus be represented, similarly to other structures, with a heap pointer transmitted on the arcs of the graph. Dennis [Den81] has generalized this representation slightly in defining an early completion data structure as a binary tree created with holes in both left and right branches; Amamiya et al [AmaHM82] use a similar definition of Lisp's **cons** for their list processing data flow machine.

These proposals are similar in some respects to the suspensions used by Friedman and Wise [FriW76] in implementing a lenient (non-strict) **cons** in a functional language. The suspension is similar in that it represents a data structure which is not yet fully elaborated; it is different in that the two cells of a suspension are not left empty (holes), but contain references to an environment which is capable of computing the required value. The suspension is demand driven (see below) in that it is coerced when its value is required, whereas an early completion data structure is data driven in that an empty component is filled as soon as the value becomes available.

An I-structure (Incremental structure) [ArvT81] is an array-like structure (specifically, an Id structure with integer selectors) with constraints on its construction and consumption to permit elements to be accessed randomly (as distinct from the sequential access required for streams) before the entire structure is complete. An I-structure must be produced in such a way that, once a value has been appended at a given selector, no other value will ever be appended at that position in the structure. If the consumer of an I-structure refers to each element at most once, selection of an element can be regarded as a destructive read operation. I-structures permit increased asynchrony in that individual appends can be made out of order (as well as accessing elements before the structure is complete). They frequently offer an advantage over streams in simplicity of coding—a program using ordinary structures may need to be extensively rewritten to use streams in improving parallelism, whereas interpreting an ordinary structure as an I-structure can offer a similar performance improvement with no re-coding. Arvind and Iannucci [ArvI83] propose the implementation of I-structure storage by associating presence bits with each memory cell, which indicate whether or not a value has been written into the cell. A cell operates similarly to a hole—if an access is attempted before a value is written, the read request is queued until the cell is written.

1.7 Demand driven computation

The data flow models described so far are data driven in the sense that operations are enabled for execution by the arrival of all operands. In a demand driven model, an operation is initiated by the arrival of a demand for its value; in computing that value, it demands values from operands as needed to perform the computation. A demand driven system is a data flow system in the sense that data requirements determine the execution of operations. It is different from a data driven system in that only values needed for the overall result are computed; a data driven program computes values whenever possible, necessitating the use of gates to discard unwanted values.

The demand driven model has its origins in reduction systems using graph reduction with an outermost computation rule [Man74, TreBH82, Tur81]. In a reduction system, a program is a series of function definitions, and an expression denoting the result of the program. The basic operation used in expressions is function application; an expression is evaluated by reducing it to a value. Evaluation commences with reduction of the program expression; a reference to another definition within the expression is seen as a demand for the value of that function application. Evaluation of the original expression is suspended while the definition is invoked and reduced. Invocation is by following a pointer to the definition, using an appropriate mechanism for parameter binding (as in the lambda calculus [Lan66b, FriW78, Abd76]); reduction of the designated expression may, of course, result in further demands. Evaluation of the original expression is resumed when the demanded expression has been reduced to a value. The advantages of this evaluation mechanism are that sub-expressions can be shared, and reduced at most once, that only necessary computations are attempted, and that infinite data structures can be manipulated if only finite portions are demanded [FriW76, KelLP79].

Reduction systems have been used as the basis of several machine architecture proposals [KelLP79, KelLP78, Ber75, DarR81, ClaGMN80]. These proposals, and their relationship to data driven computation, are discussed in [TreBH82].

An advantage of data driven computation over demand driven is that is more efficient in situations where all values are required, for example, arithmetic expression evaluation; propagation of demands constitutes an unnecessary overhead, and concurrency is reduced. Demand driven computation is, however, relevant to this thesis because it is known to provide a basis for implementing Lucid correctly according to its mathematical semantics [AshW77a]. A characteristic of the model discussed above is that demands are implicit in the evaluation mechanism. This thesis does not use such a model; in preference, a data driven model is used to explicitly express demand flow, thus making it easier to combine data and demand driven target schemes.

1.8 Implementation issues

Ashcroft and Wadge [AshW77a] distinguish three possible methods of implementation for Lucid (or a subset). The first is analysis of the program into loops, and the generation of iterative object code. Secondly, they suggest translation of a program into data flow networks of the type suggested by Kahn [Kah74, KahM77], with a network for each definition, and a node for each Lucid operator [Wad81]. There are deficiencies with these methods: redundant computation may be performed, and some legal programs cannot be translated either into loops or data flow networks. The third method, demand driven evaluation based on the formal semantics, is completely correct in that no redundant computation is attempted. A demand is a request for the value of a variable at a particular time; evaluation is initiated by requesting the result of the main program at time 0, and satisfying additional requests as they propagate.

The goal of any LX or Lucid implementation is to compute values of histories which agree with those specified by the mathematical semantics. Ideally, only those values required to determine $[\![result]\!]$ should be computed, and each such value should be computed once only. Implementations frequently fall short in either or both of these respects. In this thesis, a ***redundant computation*** is one which attempts to compute a value not required to obtain $[\![result]\!]$; a redundant computation is undesirable because it may be either erroneous or non-terminating. For example, many implementations fail to correctly implement intermittent histories, those histories which have some undefined elements, representing values which need not be computed to obtain the program result; an attempt to compute such a value is redundant at best, nonterminating at worst. The term *recomputation* refers to the computation of a value more than once; excessive recomputation should be avoided because it is inefficient.

The implementation described in Chapter 4 follows the first method of Ashcroft and Wadge in that it considers those programs which can be readily analysed into loops, hence, the language implemented is a subset of LX, rather than the full language. One version of this implementation generates imperative code. Another version generates DDF-like data flow graphs. Like method two of [AshW77a], in generating these graphs, it exploits the fact that programs are free of side effects, and impose no sequencing other than data dependencies. It is also loop based, a characteristic of method one. The graphs are built from low-level nodes, and the design of the implementation is influenced more by considerations of target language schemes than source language constructs.

One way of adding to the power of the implementation is to raise the level of abstraction of the target language. For example, a Clause Lucid **function** clause can be understood, in terms of Kahn and MacQueen networks, as requiring dynamic reconfiguration of the network to implement recursion in the clause. It is possible, with considerable effort, to define DDF-like schemes with similar capabilities, and hence implement more powerful source language features. However, such schemes still suffer from a problem fundamental to data driven networks: redundant computations may be attempted. The implementation described in Chapter 5, in striving for a correct implementation of full LX, combines Ashcroft and Wadges' methods two and three. It is based on an operational semantics which models the flow of demands (requests for values) in computing the result of the program at a particular time. The model is designed in such a way that DDF-like schemes can be derived directly from it; these schemes are the basis of this second implementation.

Several other implementations of Lucid, Lucid subsets, and Lucid-like languages have been attempted [Far77, Hof78, Hof80, Wen82, Gla82, Pil83, Car76, DenM83]. The implementations of Farah [Far77] and Hoffmann [Hof78, Hof80] both implement subsets of earlier versions of Lucid using loop analysis; [Wen82] describes an earlier version of the subset implementation presented in this thesis. Glasgow [Gla82] develops an operational semantics, and proposes an algorithm as a possible basis for an implementation. The operational semantics is formally derived from the denotational definition of Structured Lucid [AshW79b], and gives rise to an implementation scheme with iterative and recursive components, and a dynamic dependency graph used to determine which computations should be attempted.

Pilgram's implementation scheme [Pil83] also uses method two above. An imperative, message passing language is used to implement the nodes of the network. The nodes communicate using a standard protocol which ensures, by buffering some values, that recomputation of history values is avoided. However, the method requires that history values be computed in succession, and hence some redundant computations may be attempted.

Cargill [Car76] describes a demand driven interpreter for Basic Lucid. Denbaum's implementation [DenM83] compiles ANPL (essentially Clause Lucid) into coroutine based object code, which uses recursive procedure invocations to reflect the demands issued in the course of a computation, and generators [Mar80] to supervise the computation of the values of a particular history. It is not clear from Denbaum's description, however, that the implementation is capable of the dynamic reconfiguration required to implement recursion, or that no redundant computation is attempted.

CHAPTER 2

THE PROGRAMMING LANGUAGE LX

2.1 Introduction

The programming language LX was introduced in §1.2. This chapter begins with a more detailed informal description of the language, including an explanation of its semantics based on the mathematical definition given in [AshW79b]. Subsets of LX are used to explain some aspects of semantics, and in presenting operational views of the language.

If it is supposed that execution of an LX program is initiated by demands for values of the result of the program, then execution can be understood, in operational terms, by examining the propagation of such demands to the constituent definitions of the program, which in turn will initiate the computation of further values. An important contribution of this chapter is the definition of a demand driven model of the semantics of LX, which specifies precisely how, and to where, demands are propagated. The relationship of this operational semantics to the formal definition of Lucid [AshW79b] is also considered.

It is then shown how the imposition of certain syntactic and semantic restrictions results in a language LX3 with a straightforward operational interpretation in terms of loops.

2.2 A description of LX

A syntactic description of LX, in extended BNF notation, is given in Table 2.1. An LX program consists of declarative information about the free variables of the program, its inputs, and a program body which consists of a list of definitions of the variables of the program. The definitions can occur in any order, except that all variables must be declared, and the declaration of a variable must occur before its definition.

The following symbols are used in the meta-language:

::=	definition
{ X }	0 or 1 occurrences of X
[X]	0 or more occurrences of X
|	alternative
' '	used to enclose a symbol used in both LX and the meta-language

```
PROGRAM         ::=  prog IDENT
                          [ global GLOBAL{ , GLOBAL } ]
                          DEFN_LIST
                     eprog

DEFN            ::=  VAR = RHS  |  CLAUSE  |  DECLARATION

CLAUSE          ::=  define SUBJECT [ PARAM { , PARAM } ]
                          [ using IDENT{ , IDENT } ]
                          [ freezing FREEZE_LIST ]
                          DEFN_LIST
                     edefine

RHS             ::=  EXPR [ fby EXPR ]  |
                     asa EXPR then EXPR easa |
                     wvr EXPR then EXPR ewvr |
                     upon EXPR then EXPR eupon |
EXPR            ::=  if EXPR then EXPR else EXPR eif |
                     SEXPR [ REL_OP SEXPR ]
SEXPR           ::=  [ UNARY_OP ] TERM { ADD_OP TERM }
TERM            ::=  FACTOR { MUL_OP FACTOR }
FACTOR          ::=  LITERAL  |  VAR  |
                     first FACTOR  |  next FACTOR  |  not FACTOR  |
                     hd FACTOR  |  tl FACTOR  |
                     null FACTOR  |  atom FACTOR  |
                     APPLICATION  |  ( EXPR )
APPLICATION     ::=  SUBJECT [ ( EXPR { , EXPR } ) ]

DECLARATION     ::=  TYPE IDENT { , IDENT }

TYPE            ::=  SIMPLE_TYPE  |  list of SIMPLE_TYPE

SIMPLE_TYPE     ::=  int  |  real  |  char  |  bool

DEFN_LIST       ::=  DEFN { ; DEFN } [;]

FREEZE_LIST     ::=  VAR { , VAR }  |  all  |  none

GLOBAL          ::=  TYPE VAR
PARAM           ::=  TYPE VAR
VAR             ::=  IDENT
SUBJECT         ::=  IDENT

LITERAL         ::=  NUMBER  |  " CHAR "  |  BOOL_CONST  |  LIST_CONST
NUMBER          ::=  DIGITS [ . DIGITS ] [ e [ UNARY_OP ] DIGITS ]
BOOL_CONST      ::=  true  |  false
```

LIST_CONST	::=	'[' [LITERAL { , LITERAL }] ']'
DIGITS	::=	DIGIT { DIGIT }
IDENT	::=	LETTER { ALPHA_NUM }
ALPHA_NUM	::=	LETTER | DIGIT
REL_OP	::=	< | > | **ne** | **eq** | >= | <=
MUL_OP	::=	* | / | **div** | **and**
ADD_OP	::=	+ | − | **or** | :
UNARY_OP	::=	+ | −

Table 2.1. A syntactic description of LX.

2.2.1 Variables, definitions and declarations

Most LX language constructs have as their meaning a history, that is an infinite sequence of values in which each value is of the same type. In the following description, the semantics of various LX constructs is discussed in terms of the history denoted by the construct. In §1.2, the following notations were introduced, for an LX construct X: $[\![X]\!]$, "the history denoted by X", and $[\![X]\!]_t$, "the history denoted by X at time t". It will be recalled that the history denoted by X can be written:

$$[\![X]\!] \;=\; \langle\, [\![X]\!]_0,\; [\![X]\!]_1,\; [\![X]\!]_2,\; \dots\, \rangle.$$

An abbreviated notation is sometimes used, for example:

$$[\![a]\!] \;=\; \langle\, a_0,\; a_1,\; a_2,\; \dots\, \rangle.$$

A history in which all values are equal is referred to as a "constant history"; an example is:

$$[\![2]\!] \;=\; \langle\, 2, 2, 2, \dots \rangle.$$

Additional examples are given in §1.2.

A variable declaration is introduced by a type specification, which is followed by a list of identifiers. The basic types supported are **int**, **real**, **char** and **bool**, with appropriate literals and operators (see Table 2.1). A structured type can be specified as **list of** a basic type. A value of such a structured type is either ***nil***, or a list of atoms, each of which is a value of the specified basic type. Structured literals can be constructed by writing a list of values of the appropriate type, separated by commas, between square brackets. For example, the following definitions show a declaration of a structured variable, and its definition using a literal value. Note that the definitions define a constant history.

> **list of int** *a*;
> *a* = [1,5,66,77,9];

A variable is defined using a definition of the form

$$\text{VAR} = \text{RHS}$$

where the left hand side is an identifier. The right hand side, in its simplest form, is simply an expression. Other forms of the right hand side will be described later.

Consideration is now given to the formation of expressions from variables and operators. Note that not all syntactically valid expressions are legal, for they may not satisfy the type rules of the language; type checking is described below. According to the syntax definition, expressions are defined hierarchically in terms of simple expressions, terms and factors. This groups the operators of the language by precedence as follows, in descending order (the order is based on that used in Pascal)

(1) **first next not hd tl atom**
(2) * / **div and**
(3) + − **or** :
(4) **eq** <> < > >= <=
    if_then_else_eif
(5) **fby**
    asa_then_easa
    wvr_then_ewvr
    upon_then_eupon.

The operators of the fifth group are actually not defined as part of the syntactic category EXPR, but as part of RHS; however, they can be regarded, informally, as operators with syntactic restrictions on their usage.

In discussing the meaning of operators applied to variables, two sets of operators are considered, firstly, the data operators, which are defined in terms of operators associated with the data types of the language, and, secondly, the special Lucid-like operators. The arithmetic, boolean, relational and list operators, and the operator **if_then_else_eif**, are included in the first set. The operators **first**, **next** and all operators of precedence group (5) are in the second set.

Consider the expression

$$a \ \mathbf{op} \ b$$

where **op** is an operator in the first set, and a and b are variables. In the above notation:

$$[\![a \ \mathbf{op} \ b]\!] \ = \ \langle\ [\![a \ \mathbf{op} \ b]\!]_0,\ [\![a \ \mathbf{op} \ b]\!]_1,\ [\![a \ \mathbf{op} \ b]\!]_2,\ \ldots\ \rangle.$$

The components of the history are defined pointwise, thus:

$$[\![a \ \mathbf{op} \ b]\!]_i \ = \ [\![a]\!]_i \ \mathbf{op} \ [\![b]\!]_i.$$

The history yielded by the expression can then be expressed, in the abbreviated form, as:

$$[\![a \ \mathbf{op} \ b]\!] \ = \ \langle\ a_0 \ \mathbf{op} \ b_0,\ a_1 \ \mathbf{op} \ b_1,\ a_2 \ \mathbf{op} \ b_2,\ \ldots\ \rangle.$$

When either operand is an expression, the history denoted by the expression can be found by applying similar considerations to the constants, variables and operators of the expression.

The operations on structured values are defined below:

(i) selection operators

hd	a unary operator which yields the first element of the list supplied as argument; an error occurs if the list is nil.
tl	a unary operator which yields the list formed by removing the first element of the list supplied as argument; an error occurs if the argument list is nil.

(ii) construction operator

: a binary operator which yields a new list, the head of which is the left argument, and the tail, the right argument; an error occurs if the left argument is not a value, or the right argument is not a list, of the appropriate type.

(iii) predicate

null a unary operator which yields the Boolean value ***true*** if its argument is the empty list, otherwise ***false***.

(iv) comparison operators

eq a binary operator which yields the value ***true*** if its arguments are equal, where equality is as defined below.

<> a binary operator which, given arguments a and b, yields **not** (a **eq** b).

Equality is defined thus:

- two null lists are equal;
- a null list is not equal to a non-null list;
- in the case where both lists are non-null, let the lists be x and y; then

  $$x \ \mathbf{eq}\ y \quad \text{iff} \quad (\mathbf{hd}\ x\ \mathbf{eq}\ \mathbf{hd}\ y)\ \mathbf{and}\ (\mathbf{tl}\ x\ \mathbf{eq}\ \mathbf{tl}\ y)$$

  where the **eq** operation comparing list heads is that appropriate to the type of the atoms of the list.

The conditional expression

$$\mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3\ \mathbf{eif}$$

where E_1, E_2 and E_3 are expressions, yields a history which is the pointwise extension of the triadic data operator **if_then_else** (which acts on values). This operator takes three values as arguments, one of which must be of type **bool**; the remaining two values can be of any type, but each must be of the same type. If these values are p, x and y respectively, the result yielded by the data operator is x if p is ***true***, y if p is ***false***, and ***undefined*** otherwise. The value at time t in the history yielded can be written:

$$[\![\mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3\ \mathbf{eif}]\!]_t = \mathbf{if}\ [\![E_1]\!]_t\ \mathbf{then}\ [\![E_2]\!]_t\ \mathbf{else}\ [\![E_3]\!]_t\ \mathbf{eif}.$$

Consider the Lucid-like operators of the second set. The operator **first** forms a constant history from the first element of the history denoted by its argument, while **next** forms a history by removing the first value. The meanings of the operators are defined thus:

$$\begin{aligned} [\![\textbf{first } F]\!] &= \langle\, [\![F]\!]_0, [\![F]\!]_0, [\![F]\!]_0, \ldots \rangle \\ [\![\textbf{next } F]\!] &= \langle\, [\![F]\!]_1, [\![F]\!]_2, [\![F]\!]_3, \ldots \rangle \end{aligned}$$

where F is a factor. The operator **fby** (pronounced "followed by") takes two expressions as arguments, and forms a history from the first value of the first argument history, and the entire history denoted by the second argument, thus:

$$[\![E_1 \textbf{ fby } E_2]\!] = \langle\, [\![E_1]\!]_0, [\![E_2]\!]_0, [\![E_2]\!]_1, \ldots \rangle.$$

The right hand side of an LX definition can be written in one of the following three forms, in which E_1 and E_2 are expressions:

(i) **asa** E_2 **then** E_1 **easa**
(ii) **wvr** E_2 **then** E_1 **ewvr**
(iii) **upon** E_2 **then** E_1 **eupon**

The history corresponding to any of these is defined in terms of the histories denoted by the constituent expressions.

The definition of the **asa_then_easa** (pronounced "as soon as") operator is:

$$\begin{aligned} [\![\textbf{asa } b \textbf{ then } a \textbf{ easa}]\!] &= \langle\, [\![a]\!]_s, [\![a]\!]_s, [\![a]\!]_s, \ldots \rangle \\ &\qquad \text{where } s \text{ is such that} \\ &\qquad [\![b]\!]_i = \text{true}, \; i{=}s \;\text{ and} \\ &\qquad [\![b]\!]_i = \text{false}, \; \forall\, i{<}s. \\ &= \perp \quad \text{otherwise.} \end{aligned}$$

Thus, the operator can be seen as examining the history denoted by its first argument until a value *true* is found, extracting the corresponding value from its second argument history, and yielding a history which takes on this value at every point.

The **wvr_then_ewvr** (pronounced "whenever") operator is defined (see Appendix 1) using a recursive **define** clause. It can be explained by considering the following equation, which specifies a right hand side which can be substituted for the left when appropriate:

$$
\begin{aligned}
&\textbf{wvr } E_2 \textbf{ then } E_1 \textbf{ ewvr } \; = \\
&\qquad \textbf{if first } E_2 \\
&\qquad \textbf{then first } E_1 \;\; \textbf{fby} \;\; \textbf{wvr next } E_2 \textbf{ then next } E_1 \textbf{ ewvr} \\
&\qquad \textbf{else wvr next } E_2 \textbf{ then next } E_1 \textbf{ ewvr} \\
&\qquad \textbf{eif}.
\end{aligned}
$$

The overall effect is that the operator yields a history containing those values of $[\![E_1]\!]$ for which the corresponding value of $[\![E_2]\!]$ is *true*, or, in other words, it filters out those values of $[\![E_1]\!]$ for which $[\![E_2]\!]$ is *false*. To see this, the definition should be understood, in terms of the histories denoted by the operands, as examining the first item in $[\![E_2]\!]$ and, if the value is *true*, yielding a history, the first value of which is the first value of $[\![E_1]\!]$ and with the remaining values determined by applying the operator to the operands **next** E_2 and **next** E_1; the latter yields $[\![E_1]\!]$ with its first value removed. If **first** E_2 (a constant history) yields *false*, the history returned is determined by applying **wvr_then_ewvr** to **next** E_1 and **next** E_2; in effect, the first value of each of $[\![E_1]\!]$ and $[\![E_2]\!]$ is discarded. For example, consider the variables p and x, denoting histories thus:

$$
\begin{aligned}
[\![p]\!] &= \langle\, t,\ f,\ t,\ t,\ f,\ t,\ t, \ldots \rangle \\
[\![x]\!] &= \langle\, 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6, \ldots \rangle.
\end{aligned}
$$

The operator yields:

$$[\![\textbf{wvr } p \textbf{ then } x \textbf{ ewvr}]\!] = \langle\, 0, 2, 3, 5, 6, \ldots \rangle.$$

In contrast, the operator **upon_then_eupon** (pronounced "upon") can be seen as stretching out the history denoted by its second argument according to Boolean values supplied by the first argument. For example, considering the variables p and x, as defined above, the operator yields:

$$[\![\textbf{upon } p \textbf{ then } x \textbf{ eupon}]\!] = \langle\, 0, 1, 1, 2, 3, 3, 4, 5, \ldots \rangle.$$

As in the case of **wvr_then_ewvr**, this operator is defined (Appendix 1) using a

define clause, but can be understood by considering the equation:

```
upon $E_2$ then $E_1$ eupon  =
        first $E_1$  fby
        if first $E_2$
        then upon next $E_2$ then next $E_1$ eupon
        else upon next $E_2$ then $E_1$ eupon
        eif.
```

This definition is discussed in more detail in §2.3.1.1.3. The operator can be used to merge two histories [Wad81].

This completes the description of types and variable declarations, of expressions and of right hand sides, with the exception of the meaning of an APPLICATION, which is included in the discussion of the **define** clause, immediately below.

2.2.2 The define clause

The **define** clause has two major purposes in the language; it permits the user to structure a program into parameterized functions, and also permits the use of nested subcomputations. This raises issues of scope rules and the environment in which variables are defined.

The general form of the clause is as follows:

```
define SUBJECT [ PARAM { , PARAM } ]
    [ using IDENT{ , IDENT } ]
    [ freezing FREEZE_LIST ]
    DEFN_LIST
edefine.
```

An additional language rule is that the variable *result* must be declared and defined within the list of definitions making up the body of the clause, which defines a subject by identifying it with the history denoted by *result*. The **using** list includes those identifiers which are global to the clause, and the **freezing** list, those which are frozen inside the clause; freezing will be considered in detail later. The optional parameter list includes declarative information about any formal parameters.

2.2.2.1 Scope Rules

The scope of an identifier is the region of the program over which a particular declaration of the identifier is in effect. In LX, the scope of an identifier is the clause in which it is declared (a program is a special case of a clause), and does not implicitly include inner clauses. If access to an identifier is required in an inner clause, its scope can be extended explicitly by including it in the **using** list of the inner clause; the identifier is said to be "inherited" by the inner clause. Any identifier which is accessible within a clause can have its scope extended in such a way.

Identifiers declared in a clause are referred to as local identifiers of the clause, and are declared either as formal parameters of the clause or as declarations in the list of definitions which make up the clause. Global identifiers are those which appear in the **using** list of a clause, or, in the case of the program clause, the **global** list. The latter also have declarative information attached, because it is necessary to know the type of each variable which is global to the program clause.

It is not necessary for declarations of local identifiers to appear at the head of the list of definitions of a clause. The scope of identifiers which are declared part way down the list is not the entire clause, but the region extending from the point of declaration to the end of the clause. This cannot lead to ambiguities, because a global identifier must be specified in the clause heading, and it is then illegal for the same identifier to be used as a local anywhere within the clause.

2.2.2.2 The meaning of a clause

The meaning of a variable has been defined as a history, and the meaning of an expression in terms of the histories denoted by the components of the expression. It is clear that a parameterized clause, used with different actual parameters (the meaning of each of which is a history), will, in general, yield different histories. The meaning of a clause is therefore regarded as a history function, that is, a function which maps the appropriate number of histories onto a history.

The appearance of an identifier x on the right hand side of a definition is referred to as a *use* of x. The meaning of a particular use of a clause subject (with specific actual parameters) is the history determined by applying the history function (the meaning of the subject) to the histories which are the meanings of the actual parameters.

Another notion useful for explaining clauses, and the meanings of variables within them, is that of an environment. The environment of a clause is a function which associates each identifier accessible within the clause with either a history (in the case of a variable) or a history function (in the case of the subject of a **define** clause). The notion of environment considerably simplifies the explanation of what it means to freeze a variable.

The notion of *solution* can now be defined. Informally, the solution is the environment which contains the minimum amount of information consistent with the definitions of the program; it assigns values to only those history elements which are needed to satisfy the definition of *result* in the program. Ashcroft and Wadge define the solution of a program as the least environment which satisfies the definitions of the program, where "least" is defined in terms of an information ordering on the underlying data algebra [AshW76]. It can be regarded as the least fixed point of a system of mutually recursive equations, and fixed point theory used to show the existence and uniqueness of a solution for all well formed programs [AshW76]. The least environment is a mathematical device for giving meaning to a program; it gives no indication of how the solution can be computed. In this and subsequent sections, the existence of a solution is assumed, so discussion centres on properties of the solution and environments in general, rather than on computational mechanisms; §2.4 examines the mathematical semantics in relation to an operational model.

Subsequent sections (2.2.2.2.1-5) explain in some detail the mathematical semantics of clauses. It is desirable to do this, for three reasons, firstly, to establish a correspondence between Lucid semantics ([AshW79b]; summarized in Chapter 1) and LX semantics. For explanatory purposes, it is convenient to focus on two subsets of LX, termed LX1 and LX2, which parallels the treatment in [AshW79b] of the languages ULU and LUSWIM in relation to Structured Lucid (see §1.4.3). Secondly, an understanding of the mathematical semantics provides an insight into the formal roots of the language. Finally, a detailed explanation is necessary to form a basis for arguments that the operational model described in §2.3.1 correctly implements the semantic definition.

However, a simpler description should suffice on first reading. The comments which follow, in conjunction with the examples discussed in Sections 1.2 and 2.2.3, are intended to give such an explanation of the meaning of a clause.

Conceptually, a clause denotes a history function, an object which yields a history when presented with histories as arguments. Hence, a use of a clause denotes a history; this history is $[\![result]\!]$ calculated in an environment which associates the formal parameters of the clause with the histories (possibly frozen) denoted by the actual parameters.

It may be more instructive to view a clause use textually. A use of a clause can be regarded as introducing a new clause into the program, with actual parameters substituted for formals; a recursive use causes new "copies" of the same clause to appear. This view can be justified by considering the development of Lucid from Landin's lambda-calculus-based ISWIM [Lan66a, AshW79a]. Another view, applicable to programs which do not use freezing, is to regard a program as a graph, as in §2.2.3.

These views must be modified somewhat to allow for freezing. If a clause freezes a formal parameter, $[\![result]\!]$ must be determined point by point; the environment used to determine $[\![result]\!]$ at time i associates with the formal parameter a constant history formed by extracting the i^{th} value of the history denoted by the actual parameter. Using the textual view, one can imagine a copy of the clause for each time point, each substituting a different, constant, history for the formal parameter.

2.2.2.2.1 The language LX1

In this subset of LX, no freezing is permitted. This restriction is enforced syntactically by requiring that the option **freezing none** be used at the head of every clause.

In an LX1 program, the meaning of an identifier is the same everywhere in its scope, whether the identifier is used locally or globally. It will be shown in §2.2.3 that such programs may be interpreted operationally as specifying a data driven network of autonomous processes or coroutines.

2.2.2.2.2 The language LX2

The restriction for this subset is that all parameters and globals must be frozen, and is enforced by requiring that the option **freezing all** be used with every clause.

In an LX2 program, an identifier has a different interpretation inside an inner clause, within which it is global; the inner clause is regarded as defining its subject point by point, with the values of all globals being frozen at each point. Such programs have an operational interpretation in which inner clauses are subcomputations of the enclosing computation; they can be implemented as nested loops or procedures. For example, the clause *SqRt* of program *NM* in §1.2 freezes its parameter, and can be interpreted as a nested subcomputation.

2.2.2.2.3 The meaning of a clause in LX1

The meaning of a clause is defined by considering a particular use of the clause, and explaining how the corresponding history can be determined. This is done by adapting the description given in [AshW79b] of the mathematical semantics of Structured Lucid.

Suppose that the clause subject x is defined within y, and that the clause use occurs within z, as in Figure 2.1. The environment which determines the meanings of identifiers accessible within a clause y is denoted E_y. Although z is shown as distinct from x, subsequent discussion applies equally to a recursive definition of x. If x is used directly within y, E_y and E_z are the same environment.

```
prog m;
  ...
  define y ...
    ...
    define x( f0, f1, ... ,fn-1 ) using ...
      ...
    edefine {x};
    ...
    define z ... using x ...
      int b;
      ...
      b = ...x(a0, a1, ... ,an-1 )...                ...1)
      ...
    edefine {z};
    ...
  edefine {y};
  ...
eprog
```

Figure 2.1. Using a clause.

Let H be the history associated by E_z with the clause use shown in Figure 2.1. It is convenient to regard H as being formed at successive time points as H_0, H_1, H_2, and so on, by environments E_x^0, E_x^1, E_x^2, and so on, where E_x^i is the environment E_x determined at time i. For LX1 programs, E_x^i is defined such that, for any identifier a accessible within x,

$$E_x^i(a) \quad = \quad E_x(a).$$

In other words, E_x^i is independent of i when considering LX1 programs; however, it will be seen below that this is not the case for LX2 and LX programs. Thus, H can be determined pointwise; the problem of finding the meaning of the clause use in E_z has been reduced to finding $[\![result]\!]_i$ in E_x^i.

It is clearly necessary to define the environment E_x^i which determines $[\![result]\!]$. The identifiers accessible within E_x^i can be grouped into several categories, namely, formal parameters, local variables and subjects, and global variables and subjects. The histories which E_x^i associates with the formal parameters are the meanings of the corresponding actual parameters, which are defined by E_z. The meanings of local variables can be determined using methods described in §2.2.1.

The history which E^i_x associates with a global g of x is the same as the history associated by E_y with g. This follows from two properties of the language: firstly, static scoping is used, hence the meaning of a global is determined by the environment in which it is defined, and secondly, it is required in LX1 that a global have the same meaning inside as outside the clause.

The meaning of a subject use (either local or global) within x, for example $y(a)$, can be determined in the same manner as the subject use 1) in Figure 2.1, but now the role of E_z of the preceding discussion is taken by E^i_x, and that of E^i_x by E^j_y. If the clause is recursive, the clause use is $x(a'_0,\ldots,a'_{n-1})$. In this case, it may be necessary to determine, in E^i_x, the value of this new use at time i. At first glance, this may seem circular, in that determination of the value of x at time i requires the value at time i. Looking more closely, it can be seen that this is not so—in general, the actual parameter histories associated with the new use will differ from those of the original use, thus determining a new environment for the new use. In fact, no such circularity can be involved, because it is assumed that the program is well-formed, and hence has a unique solution.

The mathematical semantics of [AshW79b] offers another viewpoint of the determination of H. The solution of a program, whose existence and uniqeness have been established, gives the same n-ary history function X_n as the meaning of x in both E_y and E_z; the history H is determined by applying X_n to the meanings, in E_z, of the actual parameters, namely the histories $[\![a_0]\!]$, $[\![a_1]\!]$, $\ldots$, $[\![a_{n-1}]\!]$. The pointwise interpretation is preferred because it generalizes readily to LX2 and LX, as described below.

2.2.2.2.4 The meaning of a clause in LX2

Assume a clause use as in the previous section. The determination of E^i_x differs from above in that the histories corresponding to parameters and global identifiers are frozen at time i.

A history frozen at time i yields a constant history, thus:

$$X \text{ frozen at } i \;=\; \langle\, X_i,\, X_i,\, X_i,\, \ldots \,\rangle$$

where X is a history.

The history H, as defined by E_z, can again be defined point by point. To determine H_i, firstly establish an environment E^i_x, defined as for LX1, except that the histories which E^i_x associates with the formal parameters are determined by freezing at time i the actual parameter histories defined by E_z; the histories associated with the globals are determined by freezing the corresponding histories in E_y. In other words, E^i_x is defined thus, for an identifier a:

$$\begin{aligned} E^i_x(a) &= E_x(a) \text{ frozen at } i, && \text{if } a \text{ is a formal or global} \\ E^i_x(a) &= E_x(a) && \text{otherwise.} \end{aligned}$$

Then, as for LX1, H_i is the value at time i of the history which E^i_x associates with ***result***, thus:

$$H_i \text{ (in } E_z) \;=\; [\![\textit{result}]\!](i) \;\; \text{(in } E^i_x).$$

2.2.2.2.5 The meaning of a clause in LX

A clause in LX can specify a mixture of frozen and unfrozen formal parameters and global identifiers. It is therefore necessary to show how the definitions above can be combined to explain the meaning of a use of a clause in LX. For both LX1 and LX2, it was possible to define H, the history denoted by the clause use, point by point. A corresponding definition is now presented for LX. At time i, H_i is determined thus:

(i) establish an environment E^i_x, in which the histories associated with unfrozen formal parameters and globals are defined as for LX1, while those for frozen parameters and globals are as for LX2;

(ii) determine H_i as the value at time i of the history which E associates with ***result***, thus:

$$H_i \text{ (in } E_z) \;=\; [\![\mathit{result}]\!]_i \text{ (in } E_x^i).$$

An additional restriction on freezing is introduced when LX1 and LX2 are combined to form LX. It is apparent from Figure 2.1 that a clause subject may appear on the **using** list of a **define** clause. As indicated in §2.2.2.2.3, such a global subject has, in the absence of freezing, the same meaning as in its defining clause. However, the definition of freezing given in the previous section applies only to histories, not to history functions. Hence, it is not permitted in LX to specify that an inherited subject be frozen. This rule is not absolutely necessary, for Ashcroft and Wadge [AshW79b] show that freezing of subjects can be defined for definitions which satisfy certain other restrictions; however, the omission of this facility simplified the language considerably, at the expense of a small loss of expressiveness.

Note, however, that it can be shown from the mathematical semantics that freezing of inherited subjects makes sense for a clause with all globals frozen, as in LX2 (and also LX3, described in §2.3.2.1). In LX2, freezing of a subject f implicitly freezes any globals of f in the new scope. This permits a simple operational understanding and implementation of such freezing; if clause f with global g is inherited into h, then the frozen *value* of g is implicitly inherited into h for use with all invocations of f.

2.2.3 Examples

This section illustrates the above presentation of the semantics by discussing specific examples, including those given in §1.2. It also shows possible operational interpretations of the examples.

Consider the program *Sums* of §1.2. As no freezing is used in this program, it can be regarded as an LX1 program. The meaning of the program can be determined using the semantic description contained in §2.2.2.2.3 ("The meaning of a clause in LX1"). It is determined pointwise, first finding $[\![\mathit{result}]\!]_0$ in E_{Sums}, the environment established by *Sums*, at time 0. The environment E_{Sums} associates histories with the variables result, sum and n; because there is no freezing, E_{Sums} is equal to E^0_{Sums}. The history associated by E_{Sums} with the global n is determined by the environment of definition of n; n is external to the program, hence $[\![n]\!]$ is regarded as being established by some input medium. For a given $[\![n]\!]$, $[\![sum]\!]$ and $[\![result]\!]$ can be found which satisfy the definitions of the program; using the values given in §1.2,

$$\begin{aligned} E^0_{Sums}(n) &= \langle\, 2, 7, 3, 6, \ldots \rangle \\ E^0_{Sums}(sum) &= \langle\, 2, 9, 12, 20, \ldots \rangle \\ E^0_{Sums}(result) &= \langle\, 2, 9, 12, 20, \ldots \rangle. \end{aligned}$$

Hence, $[\![Sums]\!]_0 = 2$, and subsequent values can be found similarly.

It was mentioned previously that an LX1 program can be interpreted operationally as a network of autonomous processes. It is possible to identify an arc of such a network with a stream of values representing a history, and a node with an operator which transforms histories; for example, the node **next** discards the first value which arrives at its input, and transmits subsequent values unchanged. The program *Sums* itself can then be regarded as a very simple network of one node which transforms a history of values to a history of running sums. The node *Sums* can in turn be viewed as a network, as shown in Figure 2.2a. This operational viewpoint is described in more detail in [Wad81, Pil83], and discussed in §7.3.

The program *NM* of §1.2 contains a **define** clause. In subsequent discussion, *result* will be subscripted to indicate the associated clause use; for example, $result_{NM}$ is the variable *result* in the main program. As in the previous example, $[\![NM]\!]$ can be found by determining $[\![result_{NM}]\!]_0$, $[\![result_{NM}]\!]_1$, and so on. The environment E_{NM} associates histories with a and $result_{NM}$, and a history function with *SqRt*; as noted previously, it is convenient to consider $[\![SqRt(a)]\!]$, the history denoted by the clause use $SqRt(a)$, rather than the history function. For convenience, $[\![SqRt(a)]\!]$ is written as *SR*. Because there is no freezing in the main program, these histories are the same for E^0_{NM}, E^1_{NM}, and so on.

SR is defined as $[\![result_{SR}]\!]$ in an environment established by *SqRt* instantiated with actual parameter a. Thus, to determine SR_0, it is necessary to find $[\![result_{SR}]\!]_0$

in an environment E^0_{SR} which freezes a at time 0. One such environment is:

$$\begin{aligned} E^0_{SR}(r) &= \langle\, 4,\, 4,\, 4,\, \ldots\, \rangle \\ &= [\![a]\!]_0 \\ E^0_{SR}(x) &= \langle\, 1,\, 2.5,\, 2.05,\, 2.006,\, 2,\, \ldots\, \rangle \\ E^0_{SR}(Count) &= \langle\, 1,\, 2,\, 3,\, \ldots\, \rangle \\ E^0_{SR}(result_{SR}) &= \langle\, 4,\, 4,\, 4,\, \ldots\, \rangle. \end{aligned}$$

Subsequent values in *SR* are determined by considering environments in which $[\![r]\!]$ is associated with $[\![a]\!]$ frozen at different points; for example,

$$\begin{aligned} E^1_{SR}(r) &= \langle\, 16,\, 16,\, 16,\, \ldots\, \rangle \\ &= [\![a]\!]_1 \end{aligned}$$

The operational interpretation of *SqRt* as a nested iteration was discussed in §1.2.

Next consider the following recursive LX1 program:

```
prog U;
  int x, upn;  bool p;
  define  upn(bool a, int b);
    result = b  fby  if  first a
                         then upn(next a, next b)
                         else upn(a, next b)
                         elf
  edefine;
  p = true  fby  not p;
  x =  0  fby  x+1;
  result = upn(p, x)
eprog
```

To find $[\![U]\!]$, it is clearly necessary to determine $[\![result_U]\!]$ pointwise. Consider time 0; the environment E^0_U gives meaning to p and x thus:

$$\begin{aligned} E^0_U(p) &= \langle\, t,\, f,\, t,\, f,\, \ldots\, \rangle \\ E^0_U(x) &= \langle\, 0,\, 1,\, 2,\, ,3,\, \ldots\, \rangle. \end{aligned}$$

Consideration is now given to finding $[\![upn(p,x)]\!]$ at time 0. The abbreviation *UP* is used for $[\![upn(p,x)]\!]$. The value $[\![result_{UP}]\!]_0$ must be determined in an inner environment E^0_{UP} which associates $[\![p]\!]$ and $[\![x]\!]$ with a and b respectively. From the definition of $result_{UP}$, it can be seen that

$$[\![result_{UP}]\!]_0 = [\![b]\!]_0 = [\![x]\!]_0 = 0.$$

It is thus unnecessary to elaborate E^0_{UP} further.

In the case of E^1_U, p and x have the same meaning as before, because they are unfrozen variables of U. Similarly, E^1_U defines a and b as above. Noting that, in E^1_U,

$$\textbf{first}\ a = [\![p]\!]_0 = \mathit{true}$$

and applying the definition of **fby**, it can be seen that

$$[\![\mathit{result}_{UP}]\!]_1 \quad = \quad [\![\mathit{upn}(\textbf{next}a, \textbf{next}b)]\!]_0$$

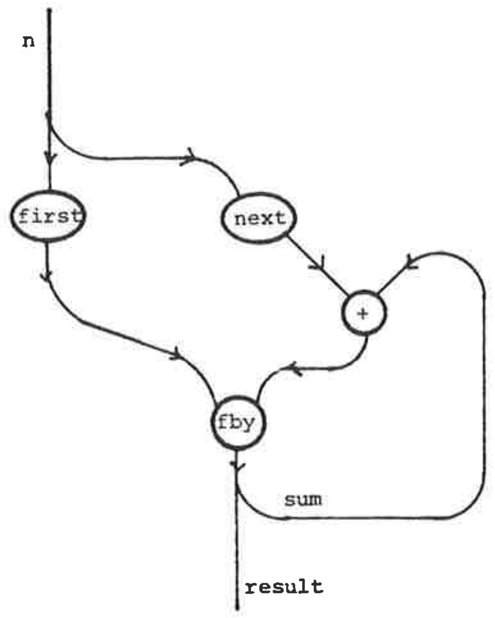
which shows that it is necessary to determine the history which E^1_{UP} associates with a recursive clause use. It is convenient to regard this use as introducing a new copy of the clause defining *upn*, which can then be instantiated with actual parameters **next** a and **next** b, in much the same way as the outer use of *upn*. Appropriate environments can then be found in the manner described above. Letting *upn_1* be the name of the copy, and *UP1* abbreviate $[\![\mathit{upn\_1}(\textbf{next}\ a, \textbf{next}\ b)]\!]$, the requirement is to determine

$$[\![\mathit{result}_{UP}]\!]_1 \quad = \quad \mathit{UP1}_0.$$

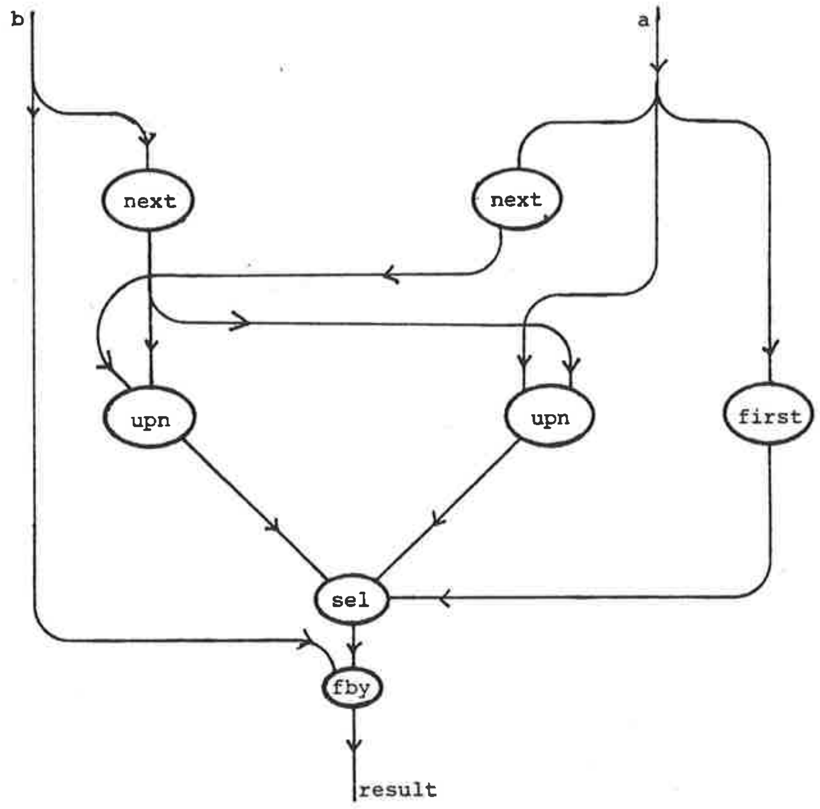
E^0_{UP1} associates $[\![\textbf{next}\ x]\!]$ with b, hence

$$\mathit{UP1}_0 = [\![\mathit{result}_{UP1}]\!]_0 = [\![b]\!]_0 = [\![\textbf{next}\ x]\!]_0 x_1 = 1.$$

The clause *upn* can be viewed as the network shown in Figure 2.2b. In Figure 2.2b, the internal node *upn* can be regarded as expanding to form a copy of the network when it is triggered for the first time; see [Wad81, Pil83].



(a) Program *Sums*



(b) Clause *upn*

Figure 2.2. LX programs as networks.

2.2.4 Discussion of language design

The design of a new language was not a primary goal of the research reported in this thesis. Rather, the intention was to take the language Lucid and perform experiments aimed at examining the practicality of implementing it in a data flow environment. Because the Lucid literature current at the time these experiments commenced did not define the language in concrete terms, [1] it was necessary to consider some aspects of language design. In particular, it was decided to examine a strongly typed variant of Lucid. The two most important design decisions are now considered in turn, firstly strong typing, and secondly clause structure.

The language LX is strongly typed; a design goal was to require static determination of the type of any expression written in the language. The principal reasons are those often put forward by proponents of strong typing in conventional languages, namely the advantages of requiring a programmer to declare the intended manner of usage of each identifier, and the ability of a compiler to check that every use of an identifier agrees with this declaration. A consequence of this decision is that histories are homogeneous—every value in the history denoted by an expression must be of the same type.

Although opportunities to write LX programs of significant size have been limited, it seems likely that the advantages and disadvantages of strong typing in LX generally will be similar to those encountered in other block structured languages. In particular, the general appearance and structure of a typical LX program bears a strong resemblance to programs in such languages. Further investigation is required before more substantive statements can be made.

Type constraints are responsible for the somewhat limited list structuring facility of LX, which permits specification of only linear lists in which a list cannot include a list as one of its elements. LX is a strongly typed language, requiring that the type of any expression be determinable statically. A problem associated with a more general

[1] This was true when an early version of LX3 was developed in 1978, but not when LX was designed in late 1982; however, the earlier design formed a satisfactory starting point for further development.

list structuring facility is that a list element can be either an atom or a list, that is, a value of either a basic type or a structured type. For example, the type of the expression **hd** x depends on the value of x. As an investigation of typing schemes was not a primary research goal, this issue has been left unresolved for LX.

Other variants of Lucid use different approaches to typing. pLucid [Fau83] requires no declaration of the type of an identifier; an identifier denotes a history of values, which need not be of the same type. Consequently, little compile time type checking is done, and histories are heterogeneous. Denbaum [Den83] requires that histories denoted by identifiers in ANPL be homogeneous, and that the type of an identifier be known at compile time, but does not require a type declaration for each identifier. Instead, only the types of the "input" identifiers (those which denote histories of values supplied externally) need be declared, and a type inference algorithm is used to deduce the type of all other identifiers.

LX uses clause oriented syntax for functions, but uses one kind of clause, in contrast to other clause oriented variants of Lucid, namely Clause Lucid and ANPL, which have several different kinds of clause. Precisely the same clause semantics can be achieved in LX as in the other languages by using the **freezing** option appropriately. The syntactic use of one kind of clause in LX thus seems a desirable simplification.

Clause oriented syntax is used for two principal reasons, firstly, clauses provide a convenient mechanism for placing the declarative information required in LX for parameters and globals. Secondly, it is felt (somewhat subjectively) that clauses encourage structuring of a program into units of a "reasonable" size.

The use of phrase oriented syntax in Structured Lucid (the **valof** phrase) and pLucid (the **where** phrase) reflects the evolution of these languages from Landin's "applicative expressions" embodied in the language ISWIM [Lan66a]. However, LX clauses can be transformed directly into Structured Lucid or pLucid phrases. Transformation from Structured Lucid to LX, and from pLucid to LX is a little more complicated because each allow "anonymous phrases" as components of expressions, and use a different syntax for specification of freezing; syntactic transformations can be defined, but a complete discussion is beyond the scope of this thesis.

2.3 Operational Views of LX

One operational view of a Lucid [AshW76, AshW77a, Pil83], and hence of an LX, program is as a network of processes or coroutines; this view was described briefly in the preceding section. It is due to Ashcroft and Wadge, and its relationship to the operational views used in this thesis is further explored in §7.3.

Another operational view identifies the definitions of a program with particular aspects of a loop [AshW77a]. An advantage of this approach is that it permits efficient implementation, but it is restrictive in that not all programs (for example, those which use the **wvr_then_ewvr** or the **upon_then_eupon** operators) can be interpreted naturally in this way. In §2.3, LX3, a further subset of LX, is defined, with restrictions which make it possible to interpret all programs in terms of loops. The implementation of LX3 is described in Chapter 4.

The operational semantics of LX presented here is capable of describing all LX programs. The semantic model is based on the notion that computation is driven by the arrival of demands for particular values in the result history of the program. It is an "information structure model" [Weg71], in which the semantics is described in terms of transformations of appropriate structures; the structures used are described below. Information structure models have also been used recently by Denbaum [Den83] to describe the semantics of the Lucid-based language ANPL.

2.3.1 A demand driven operational semantics for LX

The purpose of the semantic model is to describe the propagation of a demand through a program. The destination of a demand is regarded as being a point in the text of the program, for example, it might be a clause, a definition, or an expression. Computation is initiated by sending a demand to the program itself from an external source; a demand carries a non-negative integer n, the demand number, the receipt of which means that the value $\llbracket result \rrbracket_n$ is required. One demand must be sent for each value of $\llbracket result \rrbracket$ required.

An example is now given to illustrate the concepts of the model. Consider the program *OddNos* of §1.2, and suppose that a demand number of 1 is received by the program. The demand number 1 is propagated to the definition of *result*, thereby demanding the value $\llbracket result \rrbracket_1$, which equals $\llbracket i + 1 \rrbracket_1$.

It is clearly necessary to determine i_1, hence the demand number 1 is propagated to the definition of i, and thence to the right hand side of the definition of i. The right hand side can be viewed as an expression with the following structure

```
      fby
     /   \
    0     +
         / \
        i   2.
```

In this model, the operators of such a structure are regarded as transforming an incoming demand number, according to the nature of the operator, into one or more demand numbers, which are sent to the operands of the operator. In this case, the demand number 1 is sent to the operator **fby**. From the definition of **fby**, it can be seen that

$$\llbracket i \rrbracket_1 \quad = \quad \llbracket i + 2 \rrbracket_0$$

and the demand number 0 is propagated to the operator +. This operator acts pointwise on its operands, hence the demand is propagated unchanged to each operand, so that the value at time 0 of each of $\llbracket i \rrbracket$ and $\llbracket 2 \rrbracket$ is demanded. The former can be seen as propagation of the demand to another copy of the definition of i; a demand number of 0 arrives at the **fby** operator of this copy. It is then transmitted to the operand 0, a constant history which produces the value 0 in response to any demand. Thus, the value of $\llbracket i \rrbracket$ at time 0 is 0, and thus:

$$\llbracket i + 2 \rrbracket_0 = \llbracket i \rrbracket_0 = \llbracket 2 \rrbracket_0 = 0 + 2 = 2.$$

The value 2 is then returned to the point from which it was demanded; the return of values retraces the path of demands, and hence 2 is returned as the value $[\![i]\!]_1$, and 3 as the required value of the result of the program at time 1.

2.3.1.1 Specification of the model

The model specifies a *demand transformation* (DT) for each of the principal syntactic categories of the language, namely PROGRAM, DEFN, CLAUSE, RHS, EXPR, the various operators, and IDENT. The transformation specifies both the demands generated from the incoming demand number, and the destination of each such demand. For example, the model specifies that a program transforms an incoming demand by propagating it to the definition of *result* in the main program.

As mentioned above, the destination of a demand is regarded as being a point in the text of the program. When the text is that of a clause, it is likely that demands will be sent to the clause from several points of use in the program. To aid in distinguishing between demands arriving from different textual sources, and from recursive clause uses, the notion of an instance of a clause is introduced. An *instance* of a clause has two components, firstly, the *text* of the clause, and, secondly, an *environment table*, used to resolve uses of global variables and formal parameters.

Hence, the basic structures of the model are the demand and the environment table. A demand is defined as a pair

$$\langle\ Num,\ Inst\ \rangle$$

where *Num*, a non-negative integer, is the demand number, and *Inst* the instance through which the demand is passing. Because this model concentrates on the flow of demands, and is not concerned with the return of values, it is necessary to record only the instance in which a demand currently resides, not the history of instances through which the demand has passed.

Figure 2.3 summarises the relationships between the structures of the model. An environment table (ET) has two components. The first, *FreezeVal*, contains the time

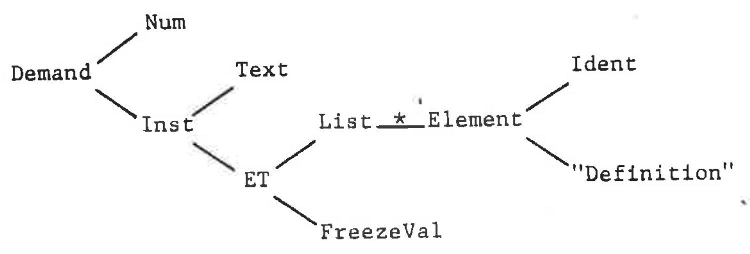


Figure 2.3. Relationships between structures of the model.

at which identifiers defined by the ET are to be frozen, if freezing is required. The second, *List*, is a list with an entry for each global and formal parameter of the clause, each entry taking the form

$$\langle \; Ident, \; User, \; NewIdent \; \rangle$$

where *Ident* is the name of the formal or global of the clause, *User* is the instance in which the use of the clause occurred, and *NewIdent* is either an introduced identifier or an identifier. *User* and *NewIdent*, taken together, identify a point (often a definition), in the text of the instance *User*, which can be used in resolving a use of *Ident*. An introduced identifier, rather than an identifier, is used when an actual parameter is an expression, *E*; in effect, it is an additional identifier, substituted for the actual parameter, and defined with *E* as its right hand side.

DTs are defined by statements in a simple, imperative language with the following Pascal-like constructs:

control primitives:	sequencing
	conditional (**if_then_else, case**)
	repetition (**for, while, repeat** and **forall**)
assignment:	:=
compound statement:	**begin** ... **end.**

The semicolon is used as a statement separator, and local variables are used as required, without declaration.

The construct

with *t* **do**

where *t* represents an ET, is used, as in Pascal, to facilitate access to the components of *t*. Dot notation may be used to access a component of a pair, and elements of the list of entries in an ET can be referenced by identifier name. For example, *t.List*[*x*] yields a ⟨*User, NewIdent*⟩ pair corresponding to the identifier *x*.

The repetition construct

forall *v* **in** *list* **do**

defines a loop in which successive elements of the list are assigned to the variable *v*. Lists of formal parameters and globals associated with a subject *x* can be referred to as *Formals(x)* and *Globals(x)* respectively. A list *Actuals* of actual parameters is also available; see §2.3.1.1.2 for details of its use.

To express the propagation of demands, the following primitive is available:

transmit D [**envof** INSTANCE] **to** DESTINATIONS [**yielding** V].

D is a demand or demand number, INSTANCE is an instance, DESTINATIONS a list of points (see below) to which the demand is propagated, V is a variable, and square brackets indicate optional components. Execution of the primitive causes the demand to be propagated to the points named, in a manner dependent on the options specified, as follows:

(i) if the **yielding** option is used, only one destination may be specified, and the value computed in response to the demand is stored in the variable specified;

(ii) if D is a demand, use of the **envof** option causes INSTANCE to replace the *instance* component of the propagated demand, otherwise the *instance* component is unchanged. Each destination is a point in the text of the instance so determined; a point can be a definition, the right hand side of a definition, or an expression;

(iii) if D is a demand number, the **envof** option must be used to specify the instance component of the demand transmitted.

The primitives *CreateInstance(subject)* and *CreateTable(inst)* can be used, respectively, to create an instance of the clause with the given subject, and return an empty ET associated with the instance *inst*.

The primitive *Class(ident)* returns the class (*local*, *global* or *formal parameter*) of an identifier, and the primitive *Table(d)* the ET corresponding to the *instance* component of the demand ***d***. The predicates *Frozen(ident)* and *Subject(ident)* indicate respectively whether or not an identifier is specified as frozen, or defined as the subject of a **define** clause.

2.3.1.1.1 Program

The DT corresponding to the syntactic category PROGRAM is shown in Figure 2.4. It is assumed that *p*, an identifier for the main program, is known to the DT; ***d*** is the incoming demand (from an external source). The DT uses two local variables, namely *p1*, which holds an instance value, and *z*, which is used as a control variable in iterating over the list of program globals. The arrival of the demand triggers the DT, to create an instance of the main program (*p1*) with an ET to service the demand. Each entry of the environment table indicates that the *User* of the program is the external instance by setting it to the special instance value *external*; the *transmit* primitive, if given such an instance specification associated with a demand, causes the demand to be propagated to the appropriate external "device" capable of handling the demand. The incoming demand, ***d***, is transmitted to the definition of *result* in the newly created instance.

2.3.1.1.2 Definition

The DT for "definition entry", given in Figure 2.5 below, must cater for both simple equations and **define** clauses. Before examining the DT in more detail, it is useful to consider the circumstances under which it is used. When an incoming demand encounters an identifier in an expression (a leaf of the expression tree introduced in §2.3.1), the demand is propagated to the definition entry associated with the identifier.

```
begin
   p1 := CreateInst( p );
   with CreateTable( p1 ) do
      begin
         FreezeVal := d.Num;
         forall z in Globals( p )
            begin
               List[z].User := external;
               List[z].NewIdent := z;
            end
      end;
   transmit d envof p1 to result
end
```

Figure 2.4. DT for PROGRAM.

Associated with the *text* component of the instance corresponding to each clause of the program, there is a definition entry for each identifier accessible within the clause. In the case of a local, the definition entry is its definition, whereas for a global or formal, it can be regarded as a unique "placeholder" for the identifier in the text of the clause. The primitive *Class* is used by the DT to distinguish between such definition entries.

In the DT, the incoming demand is referred to as d, and the identifier defined by the definition entry as x. The right hand side of the definition of a variable is identified as *rhs*. If x is the subject of a **define** clause, information about the use from which d was propagated is represented as *ui*, the instance in which the use occurred, and *Actuals*, a list of the actual parameters associated with the use; *Actuals(z)* yields identification of the expression which makes up the actual parameter corresponding to a given formal z.

If the definition entry is not that of a local identifier, the demand is redirected to a definition entry in another instance, as recorded in the ET. The demand number is changed appropriately if the identifier is frozen within the current instance. Redirection is intended to propagate the demand one step closer to an instance in which the demand can be satisfied locally; it is discussed further in §2.4.2. If x is found to be a local variable, the demand is routed to the right hand side of the definition, within the same instance.

If x is a subject, a new instance is created. The ET associated with this instance is

```
begin
  case Class(x) of
    global, formal:
      with Table( d ) do
        begin
          if   Frozen( x )
          then d.Num := FreezeVal;
          transmit d envof List[x].User
                     to List[x].NewIdent
        end;
    local:
      if not Subject(x)
      then
        transmit d to rhs
      else
        begin
          x1 := CreateInst( x );
          with CreateTable( x1 ) do
            begin
              FreezeVal := d.Num;
              forall z in Formals( x )
                begin
                  List[z].User := ui;
                  List[z].NewIdent := Actuals[z]
                end;
              forall z in Globals( x )
                begin
                  List[z].User := d.Inst;
                  List[z].NewIdent := z;
                end
            end;
          transmit d envof x1 to result
        end
  endcase
end
```

Figure 2.5. DT for a definition entry.

created with an entry for each formal parameter and global identifier of the definition. An entry for a formal parameter records the introduced identifier corresponding to the actual parameter, in the instance of use. Note that the DT is executed in the instance i in which x is defined as a local, which is not necessarily the instance in which x is used. It is assumed that necessary information about the point of use is available to the instance i, as *ui* and *Actuals*. An entry for a global records the current instance, reflecting the fact that uses of global identifiers are resolved statically. The demand d is propagated to the definition of *result* in the newly created instance.

2.3.1.1.3 Right-hand-side and expression

Both a right-hand-side and an expression consist of operators and operands; for the purposes of this model, a right-hand-side will be regarded as an expression. As indicated in the example above, an expression tree can be formed, in which the internal nodes are operators and the leaves identifiers and literals; it is assumed that all expressions are in tree form. In forming the tree, a use of a clause is regarded as a leaf, and not expressed in terms of more primitive operations; propagation of a demand to actual parameters is handled as described in the preceding section.

The simplest expression consists of a single node, either a literal or an identifier, with no operator; this case is considered later.

If the expression is not a single node, it consists of an operator, and between one and three operands, each of which is an expression. The demand is propagated to the operator, and, depending on the nature of the operator, further demands are transmitted to one or more operands. The DT corresponding to each operator of the language is specified below.

In each case, it is assumed that the incoming demand is stored in the variable d. Different notations are used in naming the operands, depending on the particular operator. For unary operators, the operand is referred to as E. For binary data operators, and the operator **fby**, the operands are named E_1 and E_2. The operands of the conditional are referred to as C (the condition), E_1 and E_2 (the arms of the conditional). For the operators **asa_then_easa**, **wvr_then_wvr** and **upon_then_eupon**, the Boolean operand is named C, and the other operand, E. Propagation of demands by operators is always within the same instance.

In the DTs which follow, local variables *d1* and *d2* are demand numbers, as distinct from demands.

The DTs for data operators, [1] and the conditional, are specified in Figure 2.6a.

[1] Note that the semantics of the operators **and** and **or** differ from Lucid, which specifies them as non-strict operators; here, it is assumed that all operands are defined. The implementation of non-strict

```
  unary data operators:  transmit d to E
 binary data operators:  transmit d to E1, E2
           conditional:  begin
                           transmit d to C yielding v;
                           if v
                           then transmit d to E1
                           else transmit d to E2
                         end
```

Figure 2.6a. DTs for data operators and conditional.

In §2.2, data operators were defined pointwise. Consequently, for all such operators, the demand is propagated unchanged to each operand. The conditional, although regarded as the pointwise extension of the triadic data operator **if_then_else**, is treated differently. It is clear that a value need be demanded from only one arm of the conditional, depending on the value of the condition. The demand d is propagated first to the condition, and then, depending on the value of the condition, to the appropriate expression; redundant computation is thus avoided.

For the special operators **first**, **next** and **fby**, the DTs are as shown in Figure 2.6b. The operator **first** yields the value at time 0 of the history denoted by its operand, hence a demand number of 0 is propagated, regardless of the value of the incoming demand number. From the definition of the operator **next**, given in the previous section, it can be shown that

$$[\![\textbf{next } F]\!]_t \quad = \quad [\![F]\!]_{t+1}$$

where F is a factor. Hence, a demand number of $d+1$ is propagated to the operand of the **next** operator. It is apparent from the definition of the operator **fby** that

$$[\![E_1 \textbf{ fby } E_2]\!]_t \quad = \quad \textbf{if } t = 0 \textbf{ then } [\![E_1]\!]_0 \textbf{ else } [\![E_2]\!]_{t-1}$$

where E_1 and E_2 are expressions. Accordingly, a demand is propagated to either the left or the right operand, depending on the value of the incoming demand number.

Consider the operator **asa_then_easa**, the DT for which is shown in Figure 2.6c.

semantics requires that computations for each operand be spawned in parallel; such a computation must be terminated if it is found that its result is not required. It may be possible to implement this in data flow systems of the type described in this thesis by passing signals through incremental parameter and result structures (described in subsequent chapters), but further investigation is required.

```
first:  transmit 0 envof d.Inst to E
next:   transmit d.Num + 1 envof d.Inst to E
fby:    if d.Num= 0
        then transmit 0 envof d.Inst to E1
        else transmit d.Num−1 envof d.Inst to E2
```

Figure 2.6b. DTs for special operators.

```
begin
  d1:= 0;
  repeat
    transmit d1 envof d.Inst
          to C yielding v;
    d1:=d1+1
  until v;
  transmit d1−1 envof d.Inst to E
end
```

Figure 2.6c. DT for operator **asa_then_easa**.

The DT demands values from the condition associated with the operator until a value *true* is obtained; the variable *d1* is used to store the associated demand number, which is propagated to the second operand. It can be seen from the DT that

$$\begin{aligned} [\![C]\!]_i &= \mathit{false},\ \forall\, i < t, \text{ and} \\ [\![C]\!]_i &= \mathit{true},\quad i = t \end{aligned}$$

where t is the demand number ($d1-1$) sent to E. Hence, the DT correctly implements the definition. The behaviour of the DT is independent of the incoming demand number, which is consistent with the fact that the history yielded by **asa_then_easa** is constant; the same value is produced in response to any demand.

The DT for the operator **wvr_then_ewvr** is given in Figure 2.6d. It was stated in the previous section that, given

$$[\![\textbf{wvr}\ C\ \textbf{then}\ E\ \textbf{ewvr}]\!] = X$$

then the history X contains those values of $[\![E]\!]$ for which the corresponding value of $[\![C]\!]$ is *true*. It is clear that

$$X_0 = [\![E]\!]_i,\ \ X_1 = [\![E]\!]_j,\ \ \ldots$$

where i is the first value such that $[\![C]\!]_i$ is *true*, j is the second such value, and so

```
begin
   d1:= 0;
   for d2:= 0 to d.Num do
      repeat
         transmit d1 envof d.Inst
                  to C yielding v;
         d1:=d1+1
      until v;
   transmit d1-1 envof d.Inst to E
end
```

Figure 2.6d. DT for operator **wvr_then_ewvr**.

on. It follows that, to find X at time t, values of $[\![C]\!]$ must be demanded until t *true* values have been found; the time, relative to $[\![C]\!]$, of this value should then be issued as a demand to E. The DT shown in Figure 2.6d implements this interpretation.

Finally, the DT for **upon_then_eupon** appears in Figure 2.6e. Consider

upon C **then** E **eupon**

which yields a history, X. The behaviour of this operator is now described, based on the definition in §2.2.2. The value of X at time 0 is produced immediately, from the expression **first** E, which yields $[\![E]\!]_0$. How the remainder of X is produced depends on $[\![$**first** $E]\!]$, in that, if it is (the constant history denoted by) *true*, the remainder of X is produced from the recursive application of the operator to the expressions **next** C and **next** C. In effect, $[\![E]\!]_0$ is released as a value of the result (X), and the histories $[\![C]\!]$ and $[\![E]\!]$ are "advanced" one step before finding the next value in X. If **first** C is *false*, the remainder of X is determined using the expressions E and **next** C. That is, $[\![E]\!]_0$ is released, as before, but only $[\![C]\!]$ is advanced before finding the next value; the same value of $[\![E]\!]$ will be released in the recursive application of the operator. The overall effect is to form X by first releasing $[\![E]\!]_0$, and then to release one further value for each value of $[\![C]\!]$; if the value of $[\![C]\!]$ is *true*, the next value of $[\![E]\!]$ is the value released, otherwise it is the same value of $[\![E]\!]$ as was previously released. From this, it follows that

$$X_t = [\![E]\!]_j, \quad t > 0$$

where j is the number of *true* values in

$[\![C]\!]_0, [\![C]\!]_1, \ldots, [\![C]\!]_t$.

```
begin
   d2:= 0;
   for d1:= 0 to d-1 do
      begin
         transmit d1 envof d.Inst
               to C yielding v;
         if v then d2:=d2+1
      end;
   transmit d2 envof d.Inst to E
end
```

Figure 2.6e. DT for operator **upon_then_eupon**.

2.3.1.1.4 Identifiers and literals

The DT's which define the propagation of a demand from a leaf node of the tree representation of an expression can now be presented. A leaf node can be either a literal or an identifier. In the case of a literal, for which the DT is

literal: the demand is not propagated

the demand is absorbed, because it can now be satisfied, and the corresponding value returned to the source of the demand, and used in some way. This model is concerned with the flow of demands, and hence does not specify in detail how values are returned and used, except when required in determining the generation of demands; similar techniques could, if desired, be used to model the return of values from literals to the source of the demand, and the transformation of values by operators. The technique used in the implementation derived from this model is described in §5.3.3.5.

The DT for an identifier x is given by

identifier: **transmit** d **to** x.

The demand d is transmitted to the definition entry, in the current instance, for the identifier. Its propagation from there is determined by the DT for a definition entry, as described above.

2.3.2 A loop based operational description of LX

An objective in the original development of Lucid was a mathematical description of iteration; Ashcroft and Wadge state [AshW77a]

> A Lucid program can be thought of as a collection of commands describing an algorithm in terms of assignments and loops; but at the same time Lucid is a strictly denotational language, and the statements of a Lucid program can be interpreted as true mathematical assertions about the results and effects of the program.

However, not all legal Lucid (and hence LX) programs can be broken down into simple loops. Ashcroft and Wadge [AshW77a] cite an example to show this; the example is given here in LX:

```
prog FACT;
   int n, result;
   n =  7 fby n−1;
   result = if n ≤ 1
              then 1
              else n * next result
              eif
eprog
```

where *result* is defined in terms of its own future, and has the history

$$\langle\ 5040, 720, 120, 24, 6, 2, 1, 1, \ldots \rangle.$$

The program cannot be translated directly to a loop which updates *n* and *result* at each iteration. It is interesting to note that the definition of *result* expresses a recursive control pattern, and can be translated into a recursive function.

By imposing restrictions on the language, it is possible to define a subset such that programs written using the subset can be understood in terms of loops. This section begins by stating restrictions imposed on LX to give the language LX3. An operational description of LX3, in terms of loops, is then presented, followed by some explanation of the restrictions imposed.

The language LX3 is similar to the language Lucid-W, implementations of which have been described in [Wen81, Wen82]. The definition and description of LX3 differ slightly from that of Lucid-W in order to relate it to the descriptions of LX and its subsets LX1 and LX2. The implementations of Lucid-W are similar to those of LX3 described in Chapter 4.

2.3.2.1 The language LX3

The following restrictions on LX define its subset LX3:

(1) **First** and **next** may only be applied to an inductive variable, at most once. In LX3, they are attributes of an inductive variable (a variable defined with a **fby** definition), and can only be used as qualifiers rather than operators.

(2) All globals and parameters must be specified as frozen.

(3) The operators **wvr_then_ewvr** and **upon_then_eupon** are omitted.

(4) The definition of *result* in a **define** clause must be an equation using the **asa** alternative on its right hand side. The definition of *result* in the program clause must be part of a loop (see §2.3.2.2 for the definition of a loop).

(5) No identifier can be defined such that a value in the corresponding history depends on a subsequent value in the history (that is, it cannot be defined in terms of its own future). This restriction is stated more precisely later, in terms of dependencies between identifiers.

(6) In defining an inductive variable, the first operand of **fby** must be a quiescent expression (essentially, an expression which yields a constant history; see the next section).

(7) Structures are omitted.

Each of these restrictions will be explained in more detail in the sequel.

2.3.2.2 An operational description of LX3

An LX3 program is composed of a set of definitions in any textual order (see the example in Figure 2.7). The program in Figure 2.7 (adapted from Ashcroft and Wadge [AshW77a]) defines 10 values of a sequence *isprime*, such that the j^{th} value in the sequence is *true* if $2j + 1$ is prime, and *false* otherwise. The computation of "primeness" of a given n, where n is $2j + 1$, is described in the body of the outermost **define** clause.

The following histories satisfy the definitions of the program shown in Figure 2.7:

```
prog PRIME;
  int n; bool stop, isprime, result;
  n =  3 fby n+2;
  stop = asa n >  20 then result easa;
  result = isprime;
  define isprime using n freezing all;
    int i;
    bool idivn,result;
    i =  2 fby i+1;
    result = asa idivn or (i * i  ≥  n) then not idivn  easa;
    define idivn using n, i freezing all;
      bool result; int m;
      m =  2 * i fby m+i;
      result = asa (m  ≥  n) then m eq n easa
    edefine
  edefine
eprog
```

Figure 2.7. Program PRIME.

$$\begin{array}{rl} [\![n]\!] & \langle\, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, \ldots \rangle \\ [\![isprime]\!] & \\ =[\![result]\!] & \langle\, t, t, t, f, t, t, f, t, t, f, \ldots \rangle \\ =[\![stop]\!] & \langle\, f, f, f, f, \ldots \rangle. \end{array}$$

In the program, the definition

$$result \;=\; isprime$$

indicates that the program returns the history $[\![isprime]\!]$ as its result. The **asa** definition of *stop* is an artificial device which serves two purposes, firstly, it specifies a termination condition for the iteration which implements the main program, thereby indicating that only a finite prefix of $[\![result]\!]$, and hence of $[\![n]\!]$ and $[\![isprime]\!]$, is required. Secondly, the use of *result* in the expression component of the **asa** definition ensures that a dependency exists between *stop* and every variable of the loop, conforming with the definition (see §2.3.2.2) of a loop in terms of dependency relationships. This example is developed further in §6.5.1.

In LX3, definitions of inductive variables provide the basis for expressing iteration. An inductive variable is defined by a recurrence relation, which can be written in the form

$$i \;=\; E_1 \;\mathbf{fby}\; E_2$$

where E_1 and E_2 are expressions. For example, the definition of n in Figure 2.7 is

$$n \;=\; 1 \;\textbf{fby}\; n{+}2$$

specifying that

$$\textbf{first}\; n \;=\; 1$$

and

$$\textbf{next}\; n \;=\; n{+}2$$

which can be understood as meaning that the initial value taken by n is 1, and, for a given iteration of the loop of which n is a part, the value of n at the next iteration is obtained by adding 2 to its current value.

When two or more inductive variables are specified in terms of each other, their definitions form part of a single loop, with each inductive variable being updated on each iteration of the loop. Execution of a loop is initiated, controlled and terminated through the definition, in an **asa** definition, of a variable, dependent on other identifiers of the loop; this variable can be regarded as the result of the loop. An **asa** definition has the form

$$a \;=\; \textbf{asa}\; C \;\textbf{then}\; E \;\textbf{easa}$$

where C is a relational expression and E an expression of the same type as the variable a. C can be regarded as specifying the termination condition for the loop implied by the identifiers upon which a depends, and E as defining the result of the loop. In Figure 2.7, the definition

$$stop \;=\; \textbf{asa}\; n \;\textbf{eq}\; 20 \;\textbf{then}\; isprime \;\textbf{easa}$$

can be understood as determining the value of *stop* (from §2.2.1, it can be seen that $[\![stop]\!]$ is a constant history) by iterating a loop involving the definitions of *stop*, n and *isprime*; the value of $[\![n]\!]$ is determined at each iteration as described above, and each value of $[\![isprime]\!]$ determined as the result of a nested iteration (see below).

The iteration continues until the current value of ***n*** is greater than 20, at which point the current value of ***isprime*** is extracted as the value of the constant history denoted by ***stop***.

If the value extracted from a loop is used in the definition of another inductive variable, then operationally the second loop will be executed after the first.

A loop can be defined as a set of definitions, constructed in the following way. The first member is an **asa** definition defining a variable, say ***a***; this definition specifies the termination condition of, and value returned from, the loop. A variable is said to depend on those identifiers which are used on the right hand side of its definition, and a clause subject depends on the identifiers specified in the using list of the clause. The definitions of those identifiers upon which ***a*** depends are added to the set of definitions constituting the loop. For a given identifier ***x*** included in the set, the definitions of the identifiers upon which ***x*** depends are also added to the set. This process continues until no more identifiers can be added.

It is often useful to distinguish three categories of variables within loops, namely "inductive", "quiescent" and "auxiliary" variables [AshW77a]. The inductive variables are those which must be updated from one iteration to the next. Informally, an expression is quiescent within a loop if textual analysis shows that it will evaluate to the same value on each iteration of the loop; a quiescent variable is one defined using a quiescent expression. The code for the evaluation of such variables can be placed outside the body of the loop. Auxiliary variables are those which have definitions expressed in terms of values known to one iteration. In Figure 2.7, ***n*** is quiescent within the definition of ***isprime***, ***isprime*** is an auxiliary variable, and ***i***, ***m*** and ***n*** are inductive.

A parameterless **define** clause provides for the nesting of loops. It is written in the form

```
define b using x,y,z freezing all;
   .....series of definitions....
   result = asa C then E easa
edefine
```

where C and E are, respectively, a conditional expression and an expression. The list of variables following the word **using** contains all of the global identifiers referenced inside the **define** clause. A nested loop is invoked once for each iteration of the outer loop, with the values of globals frozen, and thus constant during evaluation of the inner loop. The series of definitions defines *result*; at each iteration of the outer loop, the value of $[\![b]\!]$ at that iteration is determined from (the constant history) $[\![result]\!]$ in the inner loop. This form of the **define** clause is analogous to the **begin...end** notation [AshW76], and the **compute** clause [AshW77b], of Ashcroft and Wadge.

Consider a variable a defined using a conditional expression:

$$a \;=\; \textbf{if } C \textbf{ then } E_1 \textbf{ else } E_2 \textbf{ eif}.$$

In LX3, the variable a is regarded as being dependent upon all the identifiers used in C, E_1 and E_2.

In LX3, a parameterized **define** clause such as in the following example is interpreted as defining a mapping [AshW78], akin to a function in an Algol-like language.

```
define F( int x, real y ) using z freezing all;
   .....series of definitions....
   result = asa C then E easa
edefine
```

In accordance with Restriction (4), *result* must be defined in the form shown. Consider a use of such a clause in an outer iteration. Frozen values of actuals and globals are passed to the clause, determining a constant history $[\![result]\!]$, used as the value of the clause use in the outer iteration. In other words, the clause defines a pointwise function of its arguments, similarly to an LX2 **define** clause.

Ashcroft and Wadge [AshW77a, AshW76] present Lucid as a language in which assertions can be made about the histories of variables, and give general rules for expressing these assertions as equations. If Lucid is regarded as a programming language, then an implementation must be capable of coordinating the computations of individual variables in such a way that the equations are satisfied. In designing the implementation of LX3, it was considered important to find a compilation strategy which was reasonably efficient in both compilation of source language code, and in execution of the object code produced. The strategy adopted exploits the operational interpretation of Lucid in terms of loops. For this reason, it was necessary to restrict LX3 in such a way that straightforward transformations to iterative object code could be used. The basic unit of an LX3 program is the loop, and each definition in the program describes some aspect of one particular loop.

The restrictions imposed on LX3 are now discussed in this light, firstly considering language facilities for defining inductive variables. In conventional programming languages, such as Pascal, a statement of the form "i:=f(i)" can be used in a loop to determine the "next" value of *i*, that is, the value that *i* has during the next iteration, in terms of its current value. LX3 goes a little further than this simple form, in that it permits definitions to be expressed in terms of **first** and **next** values of inductive variables, as in:

```
first sum = first j
next sum = sum + next j          ...1)
```

With definitions in this form, it is possible to distinguish first, next and current attributes of an inductive variable. Thus, when we have **first** x (or **next** x), where x is a variable, the use of the operator **first** (or **next**) is regarded as qualifying the variable; however, this qualification can be made only once (Restriction (1)), so that, for example, **first next** j is not permitted. In implementing the language, this approach has the advantage that an inductive variable can be represented using a simple scheme, details of which are given in Chapter 4.

Restriction (2) states that all globals and parameters must be frozen. This permits **define** clauses to be understood as subcomputations; the outer iteration is frozen at a certain point while the computation associated with the clause proceeds, yielding a value to the current iteration of the outer loop. Subject identifiers can be inherited, for freezing of a subject is the same in LX3 as in LX2 (see §2.2.2.2.5).

The operators **wvr_then_ewvr** and **upon_then_eupon** are prohibited because they are non-pointwise. If an operation on histories is pointwise, the result of the operation at time t can be determined from the values of the operands at time t. The notion of a point in time thus corresponds naturally to an iteration of a loop, and, if all operations used in the definitions of a loop are pointwise, computations for the definitions can be grouped together and synchronized. The operators **wvr_then_ewvr** and **upon_then_eupon** cannot be easily understood in this manner. The operators **fby** and **asa_then_easa** are also non-pointwise, but, as shown above, they have special characteristics which allow a simple iterative interpretation.

A use of the subject of a **define** clause yields a single value to the current iteration of the outer loop. Hence, it is desirable that *result* be defined in the clause as a quiescent variable. The additional restriction, that *result* be defined with an **asa** definition, is imposed to permit simpler analysis of the clause into loops (see Chapter 4). In the case of a program clause, it is not necessary for *result* to be quiescent, but it is required that the definition be part of a loop. Hence, there must be an **asa** definition dependent on *result*, thus satisfying the abovementioned requirement for loop analysis. It is for this reason that the program PRIME of §2.3.2.2 defines the variable *stop*.

Restriction (5) is necessary because the values of a history are regarded as being produced on successive iterations of a loop. The computation of a particular value is always in terms of values already computed; this representation of a history does not permit access to "future" values. It will be shown in Chapter 4 that the restriction can be checked by analysing dependencies between identifiers.

Restriction (6) is imposed to simplify the loop schemes presented in Chapter 4. It means that "first" values can only be defined in terms of the "first" values of other inductive variables, or using a value returned from a loop using an **asa** definition.

Restriction (7), prohibiting the use of structures, is an implementation restriction.

It can be lifted by including support for structure operations in the implementations to be described in Chapter 4.

2.4 Relationship of the operational model to the mathematical semantics

This section is intended to answer the question of how the mathematical notion of solution of a program corresponds to the computation specified by the operational semantic model of §2.3.1. The notion of solution of a program [AshW76, AshW79b] was explored in §2.2.2.2; it is the least environment which satisfies the definitions of the program.

Consider the following analogy, which uses demand driven computation as described in §2.3.1. An environment can be thought of informally as a table, each entry of which associates an identifier with a history. Suppose that the initial demand carries demand number i, that is a request for $[\![\mathit{result}]\!]_i$. Such an initial demand can be viewed as a "probe" into an initially undefined environment associated with the program, initiating computational activity which will assign a value to the element of $[\![\mathit{result}]\!]$ which was probed, namely $[\![\mathit{result}]\!]_i$. The computational activity will cause other history elements to be probed, the flow of demands determining which ones. Thus, computation is related to the notion of environment by viewing the flow of demands as a pattern of probes into the environment.

In this thesis, it is assumed that, when $[\![\mathit{result}]\!]_i$ is computed under the demand driven strategy of §2.3.1, the following statements about the pattern of probes hold:

(i) All history elements probed are defined in the least solution, so that no attempt is made to initiate a computation not defined in the least solution.

(ii) All values probed are essential to the computation of $[\![\mathit{result}]\!]_i$.

These assumptions are supported by statements about demand driven computation in [AshW77a], and by Cargill's operational semantics of Basic Lucid [Car76]. It would, of course, be preferable to develop the analogy formally and prove the required properties, but that is beyond the scope of this thesis. Note that it is not assumed that a required history element is only probed once.

The notion of an environment as a table relating identifiers and histories is adequate for a program made up of equations only, but is inadequate if **define** clauses are used. However, the analogy can be stretched a little to cover this case. It has been established that the meaning of a use of a clause can be established in terms of three environments, namely the local environment of the clause itself, the environment of definition of the clause, and the environment in which the clause use occurs. Its meaning at time i in the environment of use is defined as $[\![result]\!]_i$ in a local environment determined by the locals of the clause, in which the meaning of a global is fixed by the environment of definition of the clause. The calculation of $[\![result]\!]_i$ can be regarded as establishing a pattern of probes into the local environment, with occasional probes into the environment of use, to determine the meaning of a formal parameter, or the environment of declaration, to find the meaning of a global.

The ET of the operational model is related directly to the notion of mathematical environment, abbreviated to ME. It can be inferred from the previous paragraph that there is a ME associated with each clause use; similarly, in the operational model, there is an instance and an ET created for each use. A ME may freeze some of its variables; an important parameter in freezing is the time i at which [result] is evaluated; the ET records this as *FreezeVal*. The "history" component of each entry in the ME is not recorded directly in the ET; rather, the DTs of the model use the ET in computing history elements which agree with those specified by the ME.

The remainder of this section demonstrates that probing is indeed performed correctly. In other words, it is shown that the demand transformations specified by the operational model satisfy (i) and (ii) above; in particular, that the operational model does not cause any unnecessary demands to be issued. Firstly, clauses with no global variables or **define** clause uses are considered; all demand propagation is within one instance. Consideration is then given to situations which require transmission of demands between instances, namely uses of global variables and **define** clauses.

2.4.1 Demand propagation within a single instance

Consider a program which uses only equations and locally declared variables. The operational model specifies that an instance of the program clause will be created on arrival of a demand, say for $[\![result]\!]_i$. All computation initiated by this demand will be carried out within the program instance.

In this case, the computation of $[\![result]\!]_i$ can be expressed in terms of the initial demand number i, and the DTs associated with the definitions and operators of the program; no additional instances or ETs need be considered. Hence, to show that no unnecessary demands are generated, it is sufficient to show firstly that the initial demand is transmitted correctly to *result*, secondly, that the operator DTs correctly transform demands, and finally that demands are transmitted correctly from a use of an identifier in an equation to the definition of the identifier.

The DT PROGRAM (Figure 2.4) specifies that the initial demand i is transmitted unchanged to the definition of *result*; this clearly agrees with the mathematical definition, which states that the value of a clause (here, the program clause) at time i is $[\![result]\!]_i$. §2.3.1.1.3 includes some discussion of each individual operator DT, which should be sufficient to show that demand numbers are propagated by the DTs only as necessary.

Consider the final requirement above. The arrival of a demand number i at a use of an identifier x is a request for the computation of $[\![x]\!]_i$. The operational model routes the demand to the definition entry for x and thence to the expression which defines x; it is clear that the propagated demand has demand number i, as required.

This explanation should convince the reader that demand propagation in this simple case is handled correctly. Subsequently, it is assumed that, once a demand for $[\![x]\!]_i$ reaches the instance of local declaration of x, the local demand propagation mechanism ensures correct computation of $[\![x]\!]_i$. Hence, in discussing demand transmission between instances, it is sufficient to establish that the demand is correctly delivered to the instance of local declaration.

2.4.2 Demand propagation between instances

Consider a demand for the value of identifier x at time i. Propagation of the demand to another instance is required if x is a local subject, a formal parameter or a global identifier. The DT "definition entry" in §2.3.1.1.2 (Figure 2.5) is used frequently in explaining this propagation, and will be referred to in this section as "DT$_{de}$".

Suppose that x is a local subject. DT$_{de}$ specifies the creation of a new instance of the clause, and transmission of the demand to *result* in that instance. The latter initiates computation of $[\![result]\!]_i$, which, for reasons mentioned in the previous section, is consistent with the mathematical semantics. Correct construction of the ET component is, of course, essential to the correct redirection of demands directed to formals and globals in the new instance; this issue is addressed in subsequent discussion.

The case alternative "global, formal" of DT$_{de}$ specifies a single step in the redirection of a demand. Since only local definition entries initiate computation, it is necessary to show that a demand is redirected to an appropriate instance of local declaration; this is shown below. It is obvious that redirection does not cause any unnecessary demands to be issued; from DT$_{de}$, it is clear that the redirection mechanism may change the components of a demand, but never generates additional demands.

The redirection mechanism is also used to model freezing of globals and formals. Three cases are now considered, namely unfrozen variables, unfrozen subjects, and frozen variables.

2.4.2.1 Unfrozen variables

Consider the propagation of the demand from its arrival at x in the current instance. It will be recalled that there is a definition entry for every identifier accessible within a clause; it is assumed that the demand is transmitted correctly to the definition entry. DT$_{de}$ specifies propagation of the demand, with demand number unchanged, to a destination recorded in the ET.

It is now necessary to examine the entries in the ET, established on creation of the current instance. Consider Figure 2.8, in which a demand for x has been propagated to i_h, an instance of the **define** clause h. Suppose that the demand originated from the use of f in line 1); clearly, the demand was transmitted through i_f and i_g, instances of f and g, before reaching x in i_h. The destination of redirection of the demand is determined by the ET of i_h, established at instance creation. As indicated above, instance creation occurs only on propagation of a demand to a local subject; hence, i_h was created by transmission of a demand from i_g, the instance of local declaration of h. Consulting DT_{de}, it is apparent that the ET entry for the global x in i_h is $\langle x, i_g, x\rangle$; hence, the demand is redirected to the definition entry for x in i_g. A similar argument can be used to show that the demand is then redirected to i_f, the instance of local declaration of x, and thence to the definition of x in i_f. This establishes that the demand is eventually propagated to the definition of x in its instance of local declaration, with the demand number unchanged. This is consistent with the mathematical definition, which makes it clear that an unfrozen variable should have the same meaning inside a clause as out.

```
define f ...
   int x ...
   define g using x ...
      ...
      define h using x ...
         ...
         int r;
         r =  ...x...;
      edefine {h};
      ...
   edefine {g};
   ...
edefine {f};
...
p =  ...f...                1)
```

Figure 2.8. A program which uses a global x.

Now consider Figure 2.9, in which x is a formal parameter of f. Suppose that a demand has propagated to e, creating instance i_e, and thence to instances i_h and i_g. Suppose further that a demand for $[\![f(a+b)]\!]$ at time i has arisen within i_g, either locally, or as a result of redirection. This demand triggers the creation of instance

i_f. Note that i_g includes a definition entry for *!ab*, an introduced identifier (§2.3.1.1) equated to the actual parameter expression $a+b$. It is now shown that the demand is propagated correctly.

```
define e ...
  ...
  int f;
  define f(x) ...
     ...
     int r;
     r =  ...x...;
  edefine {f};
  ...
  define h using f ...
     ...
     define g using f ...
        int a, b;
        ...f(a + b)...
     edefine {g};
     ...g...
  edefine {h};
  ...h...
edefine {e};
```

Figure 2.9. A program which uses a formal parameter x.

Firstly, consider $[\![f(a+b)]\!]$, abbreviated FAB, as determined by the mathematical semantics. Let E_e, E_f and E_g be the MEs associated respectively with e, f and g. It is required to find FAB_i in E_g; to do this, $[\![result_{FAB}]\!]_i$ must be determined in E_f. As there is no freezing, the meaning of the formal parameter x in E_f is $[\![a+b]\!]$ as determined in E_g, and the meanings of any globals of f are found from E_e, which is the environment of declaration of f. It follows that $[\![x]\!]_i$ in E_f is equal to $[\![!ab]\!]_i$ in E_g.

Hence, to show correct propagation of the demand beyond x, it is necessary to establish that it is redirected, with demand number unchanged, to *!ab* in i_g, the appropriate instance of g. The destination of redirection is specified by the entry for x in the ET of i_f. It can be seen, from DT_{de}, that this ET entry is $\langle x, i_g, !ab\rangle$; thus, correct redirection is achieved in one step. Note that DT_{de} assumes the availability of *ui* and *Actuals*; although the model does not make it specific, it is easy to see that this information can be attached to the propagated demand on encountering the use of f

(as is done by the implementation derived from the model; see Chapter 5).

2.4.2.2 Unfrozen global subjects

The subject f within g, as shown in Figure 2.9, is an unfrozen global. Mathematically, its meaning within g is the same as its meaning within e. The operational model is consistent with this definition if it can be shown that a demand directed to the use of f in i_g is propagated, with demand number unchanged, to i_e, the instance of local declaration of f. Arguments presented in §2.4.1 above can then be used to establish correct demand propagation to and from the identifiers of i_f.

As f is a global of g inherited from h, considerations similar to those of §2.4.2.1 can be used to show that the relevant ET entries are, in i_f, $\langle x,\ i_g,\ x\rangle$, and in i_h, $\langle x,\ i_e,\ x\rangle$. Clearly, these entries give the required redirection.

2.4.2.3 Frozen variables

Consider a demand for x at time i within i_g, an instance of a clause g in which x is declared frozen. The variable x may be either a global or a formal parameter.

Suppose that x is a global of g, that g is a local of clause f, and that $[\![g(a_1,\ldots,a_n)]\!]$ is to be found, within E_f, at time j. Mathematically, the meaning of x in E_g is specified as

$$[\![x]\!]_t \text{ in } E_g \;=\; [\![x]\!]_j \text{ in } E_f, \quad \forall\, t.$$

Assuming the correctness of demand redirection, consider the changes of demand number specified by the operational model. The computation of $[\![g(a_1,\ldots,a_n)]\!]_j$ is expressed as the transmission to the definition of g of a demand with demand number j, creating the instance i_g and its ET with *FreezeVal* set to j (see DT_{de}). It is apparent from DT_{de} that any demand redirected from x will have its demand number set to *FreezeVal*, thus satisfying the requirement established above.

Suppose now that x is a frozen parameter, and that g is invoked from a clause h. According to the mathematical semantics, assuming that $[\![g(a_1,\ldots,a_n)]\!]$ is computed at time j in E_h, the meaning of x is given by:

$$[\![x]\!]_t \text{ in } E_g \;=\; [\![a_i]\!]_j \text{ in } E_h, \quad \forall\, t$$

where a_i is the actual parameter associated with x. Clearly, the demand number of any demand redirected beyond x will be set to *FreezeVal*, which is equal to j, as required.

2.5 Discussion

The language LX differs from Lucid in several respects. The most significant of these were discussed in §2.2.4. Some possible future directions for LX are discussed in Chapter 7.

LX is a high level, nonprocedural language; the equations of a program define a solution, rather than providing a recipe for a computation. There are usually many possible ways to arrive at the solution; hence, there are many possible operational views of LX. Chapter 2 has concentrated on two of these; it provides an operational model which defines a demand driven computation of the solution, and it also describes a view which permits certain programs to be understood in terms of loops.

The operational model used is an information structure model. It is well known that such models are a valuable tool in providing abstract operational descriptions of certain aspects of computational processes [Weg71, Joh71, Mar80, Den83]. Information structure models describe computation in terms of transformations of basic structures; different models are distinguished by choice of the basic structures manipulated. In the case of LX, the mathematical definition encourages an intuitive view of computation based on propagating requirements to obtain values at certain times in certain histories, starting with $[\![result]\!]$. It is apparent that the driving force in such a notion of computation is propagation of the need to compute a value, and an important attribute of each need is the time (index) at which the value is required. Clearly, a logical choice for the basic structure of the model is a representation of a requirement, or demand, with the index of the request, the demand number, an essential component of the demand. This is the fundamental structure of the model; other structures, such as environments, are added, as components of the demand, to represent other important information.

Two principal criteria were used in designing the transformations (DTs) of the model, namely the need to describe clearly the propagation of demands in the course of computation, and at the same time maintain a close association of events in the model with the text of the program. In models of sequential computation, an abstraction of the program counter is usually stepped through the program text (for example, Johnston's "ip" [Joh71]), and events in the model thereby associated easily with specific points in the text. An LX program is not sequential, but points in the text can be seen to influence demands in different ways; an identifier affects direction of propagation, whereas an operator affects both direction and demand number. The model therefore associates a structure transformation with each syntactic category of the language, providing a direct relationship between syntactic and semantic definitions.

The model accurately describes demand driven computation in LX-specific operational terms. It has several desirable characteristics: it is concise and abstract, it describes the full language LX, it has been shown to agree with the mathematical semantics, and it is tailored to data flow concepts.

The latter suggests that the model can provide the basis of a translator to data flow graphs; the construction of such an implementation is described in Chapter 5. This is the principal application of the model in this thesis. The model could be extended to describe other aspects of computation; for example, it is suggested in §5.4 that certain DTs in the model could be used to trigger events in a model of storage management.

It is not necessary to confine the model to LX. Its essential elements are demands, and descriptions of their propagation. It should be possible to adapt these elements to many situations where precise description of demand flow is relevant. For example, in Chapter 7 it will be argued that input/output in data flow inevitably involves some notion of demand, but does not necessarily require all computation to be demand driven. A variant of this model could be used to specify such limited demand propagation.

The notion of demand used in the model, while specifically tailored to LX, is similar in many respects to that used in other demand driven models of computation. Typically, demand driven computation is identified with graph reduction [TreBH82]; an operator coerces, or demands, the computation of its operands, reducing them to values. The invocation of such a computation can be seen as the propagation of a demand, which is thus identified with a recursive function invocation and return control pattern. The model of LX extends the facilities available for controlling demand propagation; the **transmit** primitive separates the invocation of a computation and the return of a value, and several DTs (for example, that for **asa_then_easa**) implement quite complex demand transformations. In short, the model transforms demands rather than merely propagating them, and a DT may transmit several demands to obtain one value.

The language LX3 stems from a very different operational interpretation of LX. Initially, a decision is made to interpret programs in terms of loops, and the language is restricted so that it is only possible to write programs which can be interpreted in this way. In designing such a language, the restrictions must be considered carefully to permit a suitable balance between expressiveness and implementability.

The loop based operational interpretation compromises the mathematical definition to some extent. It is required that all histories be computed in order of increasing index, and that all values of a history up to the current index are computed in case they are needed in subsequent iterations. Occasionally, values are computed which would not be required in a demand driven computation. If this fact is accepted, LX3 can be seen as a different language for describing iterative computations (of course, it permits function definition and use as well). It will be demonstrated in Chapter 4 that it can be implemented as such on both data flow and conventional von Neumann machines.

CHAPTER 3

AN ABSTRACT DATA FLOW INTERPRETER

3.1 Introduction

One deficiency of the von Neumann model of computation is that it is difficult to specify and utilize parallelism. The data flow model has been proposed as an alternative which alleviates this problem. The basic principle of the data flow model is that an instruction can execute as soon as the input values required by that instruction are available; the parallel execution of many such instructions is implicit in the model.

Subsequent chapters of this thesis are concerned with the translation of LX to data flow graphs. In this chapter, the data flow interpreter, used in each of the implementations, is described. The next section describes the model on which the interpreter is based. A description of the interpreter itself is then presented; the chapter concludes with a discussion of the relationship of the model presented to other data flow models.

An example of the execution of a data flow program is given in Appendix 2.

3.2 The data flow model

The data flow model used in this thesis is essentially that presented by Dennis [Den81]. It is used because it provides an abstract description of a data driven computation, at a level which does not require consideration of machine details, yet it models most of the characteristics of data flow computers which are important in implementing a higher level language. For example, it permits the construction of graphs which use the basic operations common to all data flow schemes (data and structure operations, and control operations), but does not express lower level details such as generation of acknowledge signals [BroM79] or label manipulation [GurWG80]. In addition, it uses non-strict data construction operations, a concept used in many recent data flow proposals to improve the efficiency of data structure manipulation and to increase parallelism [Arv80, ArvI83, CalDP83].

In this thesis, Dennis' model [Den81], which uses only acyclic graphs, has been extended with additional operation codes which permit the simulation of cyclic schemes using tail recursion. This is done, firstly, because the translation schemes for LX3 presented in Chapter 4 are most conveniently expressed in cyclic form, and, secondly, to permit comparison with other data flow models which include cyclic schemes.

A data flow computation is viewed [Den81] in terms of transitions between configurations, where a configuration consists of a state S and a set of activities A. The set of activities models the set of instructions enabled for execution. The state is represented by a heap whose elements are function templates, function activations, and data structures, all of which are described below. Given a configuration $\langle S, A\rangle$, a successor configuration $\langle S', A'\rangle$ can be found by applying the function

$$\mathit{Interp}\colon\ \mathit{State} * \mathit{Activity} \mapsto \mathit{State} * \text{set of } \mathit{Activities}.$$

The function *Interp* describes the execution of some activity a by defining a new state, and a new set of activities, which replaces a in the original set of activities, A. Thus:

$$\begin{aligned} S' &= \mathit{Interp}_1(S,a) \\ A' &= \mathit{Interp}_2(S,a) \bigcup (A-\{a\}) \end{aligned}$$

where a is an activity selected arbitrarily from A; the notation Interp_1 means "the first component (the *State*) of the pair returned when *Interp* is applied to S and a". In summary, a state transition is performed by removing an activity from the set of activities, and performing the instruction defined; this will generally involve the creation of further activities, and changes to the heap. The Pascal program described later is an implementation of the function *Interp*, with appropriate representations of *State* and *Activity*, as described below.

It is assumed that a data flow program is written as a series of function definitions, each of which can be expressed in graphical form. The nodes of a graph correspond to the instructions of the program, and are numbered consecutively, as are the positions of the associated input and output arcs, corresponding to operands and results respectively. An end point of an arc is thus specified as a *link*, which is a pair *instr.posn* identifying an operand position of a given instruction; for example, 5.1 denotes the first operand (either input or output, depending on the context in which it is used) of instruction 5. An arc is represented by including a link identifying its consumer instruction at the appropriate output position of the producer instruction, and, redundantly, by the presence of its producer link among the inputs to the consumer instruction. A value can be of type integer, real, boolean, character, or a binary tree. There are instructions for performing arithmetic and boolean operations, for controlling the flow of values within an executing program, for producing constant values (the CONSTANT instruction) and for replicating values (the IDENT instruction). Operations on binary trees are supported. Instructions for function activation are also provided. Examples of data flow graphs and their execution are given in Appendix 2.

As mentioned in §1.5, the firing rule is that an instruction is enabled when a value is available on each input position; an instruction fires, or executes, some time after becoming enabled, absorbing a value at each input position and producing a value at each output position. In the case of the CONSTANT instruction, the input serves as a trigger to produce the value specified by the instruction.

The control instructions of the model are TGATE, FGATE, SWITCH and MERGE. The firing rules for TGATE and FGATE have been given in §1.5. The SWITCH instruction has identical inputs and enabling condition to the gate instructions, but transmits the data value to either the T or F output arc, depending on the value of the control input. The MERGE instruction used differs from that of §1.5; it has two inputs, and fires on arrival of either input, transmitting the value on the output arc. It is potentially non-deterministic, but in this thesis the manner of its use usually ensures that its behaviour is deterministic. In fact, it is better thought of as identifying a shared input arc used in a disciplined fashion, rather than as an instruction.

A graph, constructed from the instructions described above, is represented in the data flow model as a *function template* (FT), which is essentially an array of instructions. Both templates and function activations reside on the heap; each heap object is identified by a unique identifier, or *Uid*. A *function activation* consists of the Uid of a function template and space for the operand values of instructions. An enabled instruction is called an *activity*.

The APPLY instruction creates a new activation on the heap from a function template. Its inputs are the Uid of the FT to be activated, and actual parameter values. The output link specifies the destination of the value returned from the activation. A result is returned via a RETURN instruction in the FT; on activation of the FT, the address of the APPLY instruction is transmitted to the RETURN instruction, which, when executed, uses the destination address contained within the APPLY instruction.

Iteration is expressed as tail recursion, using the instructions IAPPLY, INCR and RETURN. A loop is similar to a function in that it produces a result in response to the arrival of a set of input values, the initial values of the loop variables. Hence, a loop is represented as a separate function template, activated by execution of an IAPPLY instruction with the initial values of the loop variables as parameters. The INCR instruction, in a loop, uses the same inputs as the IAPPLY instruction which initiated the loop, and creates a new activation for a new iteration of the loop, using updated values of the loop variables as parameters. A RETURN instruction is used to transmit the result of the loop. It is executed only once for each loop, by the activation representing the final iteration. The value is returned directly to the activation which initiated the loop, rather than through all activations; to make this possible, the INCR instruction passes on the address of the original IAPPLY instruction at the activation of each new iteration. A more detailed definition of a loop in the context of this thesis is given in §4.4.2.

The progress of a computation is modelled by changes to the heap. At the commencement of execution, the heap contains all the function templates for the program, and an activation of the "outermost" function of the program. No further function templates will appear on the heap, because new functions, as distinct from function activations, cannot be created dynamically. A function application or loop invocation results in the creation of a new activation on the heap, and return of results from an activation causes deletion of that activation from the heap. Execution of other activities causes transmission of values within an activation.

The model also includes operations for the creation and manipulation of binary trees. The operations are defined to give tree structures an "early completion" semantics, for example, elements can be selected from the tree before its construction is complete; if the element selected has not yet been computed, the selection operation is deferred until the element has been computed. Dennis points out that parameters can be passed to a function in an ECDS; because the function can access parameters before the structure is complete, function evaluation can commence with the arrival of any parameter value, rather than waiting for all to arrive. This can give a significant increase in parallelism of function activations, when compared to an APPLY operation which waits for all its operands to arrive before commencing execution of the applied function.

A binary tree consists of two components, called l and r. A component contains either a value, or a queue listing the destinations of those instructions which have attempted to select a tree component before it has been produced. The ECDS construction operators are PAIR, MKL and MKR. The first creates an ECDS in which both components are empty queues; each structure value is represented as a separate node on the heap. The operators MKL and MKR each replace a queue with a value, and forward the value to all destinations on the queue. The selection operators are L and R, which select the appropriate component if it is a value; if not, the destination links of the instruction are appended to the queue.

3.3 The data flow interpreter

This section describes in outline the structure of a Pascal program which interprets data flow programs based on the model of computation given in the previous section. The main purpose of the interpreter is to test data flow programs produced by the LX and LX3 translators described in succeeding Chapters.

The interpreter reads a data flow program from one or more sources, and then executes the program. The next sections describe the input of programs and data to the interpreter; the data structures used to represent the program and the heap, and the simulated execution of the data flow program, are then described. Some example programs, with a trace of their execution by the interpreter, are presented in Appendix 2.

3.3.1 Loading programs

A data flow program is a series of function templates, where each function template is written in the form described in §3.2. Examples are given in Appendix 2, and in Figure 3.1 below. The interpreter performs very little processing of the program text, simply converting it into a suitable internal form, which is essentially an array of instructions, with each array element containing information derived from a single line of the program text; it is described in more detail below.

The translators described in subsequent chapters produce data flow programs in the internal form, rather than as text. Consequently, the interpreter loads programs in this form as well. It is also possible to load libraries of routines from several files.

3.3.2 Input of data

Two new instructions are used, namely START and FIN, each of which have one input and one output operand. The START instruction is a means by which values are communicated to the data flow program from the sequential environment in which the interpreter runs. Although it is triggered in the usual way by the arrival of values at its input, it ignores this input and prompts for, and accepts, a new input value from the terminal, which is then transmitted as the output of the instruction. The START instruction, because it has the side effect of accepting a value from the terminal, creates an implicit data path from outside the data flow program to the activity.

It has been found convenient to submit programs to the interpreter with a more or less standard driver program. Figure 3.1 shows the START and FIN instructions as used in the driver developed for testing the factorial program described in Appendix 2, providing a mechanism with which a function can be repeatedly invoked, accepting input from, and displaying results at, a terminal. The program can be regarded as driven by data at its input if the data is stored in a file, and the START instruction modified to read from that file.

```
*0                                          Program driver
0  Ident      0.0       :  2.1
1  Return     5.1       :
2  Start      0.1       :  3.2
3  Apply      =1   2.1  :  4.1
4  Ident      3.1       :  6.1 5.2
5  Apply      =0   4.2  :  1.1
6  Fin        4.1       :
7  End        0
     Uid 0 - program driver FT
     Uid 1 - factorial FT
```

Figure 3.1. A program driver.

If used indiscriminately, such a START instruction would compromise the data driven character of the program. However, when used in the manner shown, it provides a convenient means of admitting values to a data flow program as they are required, simulating a queue of values at the input position. The means by which a similar effect can be achieved in an "ideal" data flow environment will be discussed in section 7.4. The FIN instruction displays its input operand on the terminal, and absorbs it; it is used here to simulate a sequence of values produced by the program.

3.3.3 Data structures

Figure 3.2 shows Pascal declarations of some of the principal data structures used by the interpreter. The declarations are derived from those given in [Den81].

```
StateTyp =
  record
    Heap: HeapTyp;
    NextUid: HeapSize;                 { HeapSize is suitable integer subrange}
  end;
 HeapTyp = array [HeapSize] of Node;
 NodeTyp = (Ac, Fn, St);
 Node =
  record
    case NTyp: NodeTyp of
      Ac: (Actvn: Activation);
      Fn: ( Func: Funtion);
      St: ( Str: Structure);
  end;
 Funtion =
  record
    NoInstrs: InstrNo;                 { InstrNo is suitable integer subrange}
    Fn: array [InstrNo] of Instr;
  end;
Instr =
  record
    Op: OpType;                        { OpType enumerates all op. codes}
    Constants: ValueArray;             { any constant input operands}
    NumIn,NumOut: Prange;              { no. of inputs, outputs}
    InCount: Prange;                   { no. of inputs reqd = NumIn - no. constants}
    Tgt: LinkArray;                    { each element is a destination link}
  end;
 Instance =
  record
    Opnds: Operands;                   { an array of values, one for each input}
    NoIn: Prange;                      { decremented as operands arrive}
    Active: boolean;                   { has instruction been activated?}
  end;
 Activation =
  record
    Iter: integer;
    Instances: array [InstrNo] of Instance;
    FuncUid: HeapSize;
  end;
```

Figure 3.2. Some data structures of the interpreter.

The state of the data flow computation is recorded in a variable of record type *StateTyp*. The component *State.NextUid* contains the next available Uid, a Uid being represented by an integer value, incremented whenever a new heap node is acquired. The state component *State.Heap* is an array of heap nodes. Ideally, the heap array should be dynamic, but a reasonably large static array, combined with some simple heap management techniques, has proved adequate for testing the LX implementations.

A heap node is either a function template (*Func*), a function activation (*Actvn*), or a structure (*Str*). The component *Func* implements a function template as an array of instructions, where an instruction is a structured value containing an operation code and other relevant information, as shown in Figure 3.2.

An activation is created whenever a function template is invoked. A function template records the static aspect of a data flow program, whereas an activation contains information pertaining to the dynamic aspect of a specific function invocation, namely the Uid of the template from which the activation was created, an iteration number, used for activations which represent loop iterations, and an array of *Instances*, isomorphic to the array of instructions of the template. Each instance contains those input values which have already arrived, and a count of the number of inputs which have yet to arrive before the instruction instance becomes enabled. Space is reserved for one incoming value at each input position; because the data flow graph of the function is acyclic, and a new activation is created for each invocation, at most one value can arrive at any input, so there is thus no need for queue of values on an input arc.

An instruction instance is enabled when all required input operands have arrived; as mentioned previously, an enabled instruction is termed an activity. All current activities are recorded on a list, *EnabledList*, from which activities are selected for execution. Each element of the list records an activation Uid and an instruction number.

The interpreter supports integer, boolean and structured values. A structured value is represented as the Uid of a heap node. The node contains two components, l and r, each of which is either a value or a queue. Each queue element is, in effect, an interrupted structure selection activity, which, having attempted to access an undefined structure component, is placed on the queue associated with that element until the element becomes defined by a structure construction activity. As part of the execution of such a construction activity, each interrupted activity is removed from the queue, and its execution completed by transmitting the value to the destinations specified by the suspended activity.

3.3.4 The initial state

It was stated above that a data flow computation is viewed in terms of transitions between configurations, each of which consists of a state S and a set of activities A. In the interpreter, the former is represented by the variable *State*, and the latter by the list of activities, *EnabledList*.

It is clearly necessary to define an initial configuration. Firstly, consider the heap component of the state. The function templates used in the computation are supplied to the interpreter, and entered on the initial heap. Structures are represented as heap nodes, so any structure constants required are also entered on the initial heap.

At least one activation node must also be present on the initial heap to start a computation. The interpreter prompts the user for identification of a function template to be activated initially; alternatively, it may be convenient to adopt the convention of initially activating the template at heap node 0.

The selection of initial activities from the initial activation could be carried out by finding all instruction instances with all inputs available, and entering these instances in the initial set of activities, *EnabledList*. To avoid making this search, the interpreter assumes that instruction 0 of the initial activation is the only such instance, and enters it on the activity list. It is then necessary to ensure that each function graph is constructed such that all operations which would normally require no input arcs are, in fact, triggered in some way, usually by the arrival of an output of operation 0. The initial configuration is thus defined, and execution can commence.

3.3.5 Program execution

3.3.5.1 The execution cycle

The basic execution cycle can be described thus:

```
while EnabledList is not empty do
  begin
    a <- an activity removed from EnabledList;           ...(1)
    Inputs <- input operands of a;                       ...(2)
    Execute( a, Inputs, Outputs );                       ...(3)
    transmit each value in Outputs to its destination;   ...(4)
  end.
```

Inputs and *Outputs* are temporary storage arrays, holding, respectively, the input values required by the activity, and the output values produced on execution of the activity; *Outputs*[*i*] is the value produced at output *i* of the activity.

Steps (1) and (2) of the cycle are straightforward, representing the selection of an activity for execution, and retrieval of its operands. Step (3) describes the execution of activity *a* as mapping *Inputs* to *Outputs*; this step also causes changes in the state of the computation. At step (4), output values are moved to the instruction instances specified by the output links of *a*; any destination instructions which become enabled are added to *EnabledList*, the list of activities. Steps (3) and (4) are now considered in more detail.

3.3.5.2 The execution of an activity

Activities can be divided into two categories, according to their effect on the heap; firstly, there are those whose effect on the heap is quite local as they cause values to be transmitted to instruction instances within the same activation, thus affecting only one heap node, and secondly, there are activities which can, in addition, change the structure of the heap by causing nodes to be created or deleted. Arithmetic, character, boolean and control operations are in the first category. The instruction APPLY causes an activation node to be added to the heap, and is thus in the second category. The operations RETURN, IAPPLY, and INCR, and the structure construction operation PAIR, are also in the second category.

Figure 3.3 shows extracts from the interpreter which illustrate the interpretation of typical activities in the first category. The examples are part of a Pascal **case** statement which includes one case for each operation code. The integer addition operation has two input operands, and one output. It is interpreted by accessing the input values, and computing the output value, as integers. For the operation SWITCH, the first input is used as a boolean, and used in deciding to which output the second (data) input should be sent.

```
Plus:
 With Outputs[1] do
   Begin                      { a value is stored in a variant record;}
     VTyp := int;             { VTyp is the tag, iv an integer variant}
     iv := Inputs[1].iv + Inputs[2].iv;
   End;
Switch:
 Begin
   If Inputs[1].bv then J:=1 else J:=2;
   Outputs[J] := Inputs[2];
 End;
```

Figure 3.3. Examples of interpretation of simple activities.

Figure 3.4 shows the interpretation of two activities in the second category, namely APPLY and PAIR. Three phases in the execution of an APPLY operation are shown. The inputs of APPLY are the Uid of the function template, and the parameter values to be passed to the invocation. In the first phase, an activation node is created on the heap from the template Uid. In the second phase, parameter values are transferred to the newly created activation, as follows. If n is the number of parameters, then, by convention, the first n instructions of each function template are IDENT operations responsible for distributing parameter values to their points of use in the function. Each parameter value is moved to the input operand of the appropriate IDENT instruction. Each IDENT instruction thus becomes enabled, and is added to the activity list. The third phase, *FixRA*, causes the address of the invoking APPLY instruction to be included in the RETURN instruction instance of the activation as a "return address", to facilitate transmission of function results to the invoking activation when the RETURN instruction is executed.

```
Apply:
  begin
    CreateActivation( Inputs[1] );
    ActivatePars( Inputs );
    FixRA;
  end;
Pair:
  begin
    with TempElt do              { TempElt is an empty queue }
      begin
        ElTyp := que;
        q := nil;
      end;
    with TempStr do              { each component of TempStr}
      begin                      { is an empty queue}
        l := TempElt;
        r := TempElt;
      end;
    AddStToHeap( State.Heap, TempStr );       { insert on heap,}
    UidSt := State.NextUid - 1;               { at node UidSt}
    for J:= 1 to Instruction.NumOut do
      begin
        with Outputs[J] do
          begin
            VTyp := str;             { tagof value, a}
            strv := UidSt;           { structure variant}
          end;
      end;
  end;
```

Figure 3.4. Examples of interpretation of node-creating activities.

The PAIR instruction was specified above as producing a structure in which each component is an empty queue. In the case PAIR of Figure 3.4, the first three statements show the creation of such a structure value, and its insertion on the heap as a new node. The **for** statement places a copy of the output value at each output position of the instruction. The value itself is identified by its Uid; its Pascal representation includes a tag to indicate that it is a structure.

3.3.5.3 The transmission of outputs

It will be recalled that each instruction in a function template specifies the number of outputs produced, and the destination of each. Each instruction instance also contains space for its input operands, and includes a count, *NoIn*, of the input values yet to arrive. The final phase in the execution of an activity *a* is the transmission of outputs to destinations, which proceeds as shown in Figure 3.5.

```
for j := 1 to number of outputs of a do
  begin
    i := instruction component of jth output link;
    p := position component of jth output link;
    with instruction instance i do
      begin
        copy Outputs[j] to input operand p;               ...(1)
        decrement NoIn by 1;
        if NoIn = 0
        then insert instance i onto EnabledList;          ...(2)
      end
  end
```

Figure 3.5. Transmission of outputs.

The statement labelled (1) in Figure 3.5 models a value flowing along an arc of the data flow graph. Statement (2) implements the construction of the set of activities (mentioned in §3.2) which replace *a* in the succeeding configuration.

3.4 Discussion

3.4.1 Comments on the data flow model

The data flow model described in this Chapter uses acyclic graphs to define a set of function templates which make up a data flow program. The principal advantage of the model for this research is its simplicity, which is a consequence of its intended use in a "semantic model for an experimental computer system" [Den81]. The factors which contribute to this simplicity are now considered.

Because graphs are acyclic, repetitive computation must be expressed using recursion. This encourages the decomposition of programs into comparatively small recursive function definitions, a process encouraged by the provision of function templates in the model. In [Den81], it is the textual view of an APPL program as recursive function definitions which is of most interest; in this thesis, the more important view is of the graphical function templates. Execution of such graphs is conceptually simple; a graph is activated by the arrival of values at its inputs, data flows (in one direction) through the graph, causing the execution of operations as operands become available. An operation is either primitive, or a function invocation, which causes a new activation to be spawned, conceptually by taking a copy of the graph of the function. In other words, the view of computation is one of activations coming into existence as required, and disappearing when they have fulfilled their purpose; in the model, an execution of any particular operation in a particular activation of a graph can be seen as a unique activity. It was demonstrated that building and designing an interpreter for such graphs is quite straightforward.

Another simplifying property of acyclic graphs is that the arcs transmit at most one value; the producer operation of an arc either fires once, or it does not fire at all. Consequently, an activation was implemented simply by allocating a single storage location for each arc in the graph.

3.4.1.1 Relationship to cyclic schemes

In this thesis, it was convenient to use a model which provided a concise, abstract model of data flow computation, rather than a model which is arguably more closely related to machine concepts; the emphasis is on translation to data flow at the schematic level. However, a significant aspect of Chapter 4 is the development of cyclic schemes for the translation of LX3. The question of reconciling this with a model which is fundamentally acyclic is best answered by considering relationships between the model and those data flow models which permit cyclic graphs.

In a cyclic model, arcs are regarded as paths along which a sequence of values can flow, and operations as "stations" which process streams of values. The semantics of data flow operations given by Arvind et al in [ArvGP78] expresses this approach more formally. In a sense, operations can be regarded as more permanent objects; they continue to exist while processing many values, whereas in the acyclic model an activity is transitory, and processes only one set of operands. In the cyclic model, it is convenient to express iteration by permitting an arc to "cycle back", as in the iterative schemes of Chapter 4. Such models form the basis of promising efforts to develop data flow machines [Den80, WatG82, ArvGP78]; they seem generally well accepted as models suitable as a basis for data flow hardware.

It has been shown that cyclic schemes have representations in the model of this Chapter, by introducing special operations which, although using tail recursion, permit iteration to be expressed with a scheme which is almost the same as an equivalent cyclic scheme (see Chapter 4 for complete details of such schemes). In other words, at a schematic level, the differences are very slight. Moreover, it is argued below that schemes could be developed for the transformation of the abstract graphs suitable for the interpreter described here into programs for a particular real machine.

3.4.2 Relationship of the model to data flow machines

In this section, consideration is given as to how the basic concepts of the abstract model described in this Chapter might be identified with characteristics of proposed data flow machines. It is hoped that this will give some insight into the place of the model in the spectrum between data flow schemas [DenFL74] and hardware considerations [DenM75].

The basic notions of the model are of state, activity and configuration transition. The heap represents the significant components of the state, namely function templates, activations and structures. Function templates provide storage for the instructions of a program, in much the same way as the node store of the Manchester machine [WatG82] or the activity store of the cell block architecture [Den80].

An activation represents a particular function invocation; it contains the Uid of the function template, and storage for all input operands of that template. This is very similar in concept to the instruction cell of some MIT proposals [Den80]. An activation groups together all the operand values used by a particular function invocation; by contrast, there is no such grouping in the Manchester machine. Values relevant to a particular activation are identified by a common activation label, but are stored at any position in either the token queue or the matching store. The Id proposal uses a similar concept: a value belonging to an activation has four fields associated with it, termed *u*, *c*, *s* and *i*. The field *u* is the context, analogous to the activation Uid; *c* is the code block address, comparable with the function template Uid; *s* is the instruction number.

Field i is the iteration number, used to match tokens belonging to same iteration. A similar field is included for convenience in an activation node, but as each iteration requires a separate activation, it does not have the same significance as in the Id or Manchester models.

An activity is an enabled instruction, and is represented by the Uid of an activation with an instruction number. This corresponds to an operation packet in MIT architectures [Den80]. In the Manchester machine, matching tokens leaving the matching unit cause the node store to be accessed to obtain the instruction code and destination(s), and are combined with this information to form a package which represents an activity.

Configuration transitions occur when an activity is executed; all data flow machines have processing units which perform this function. New activities are formed as a consequence of the execution of an activity; in the cell block architecture, result values are routed to destination instruction cells, whereas in the Manchester machine new values with associated activation identification are added to the token queue.

It is with structures that most difficulty arises in identifying the concept in the model with machine proposals. Consider arrays as an example. In the model, an array may be represented abstractly as a tree structure comprising a number of heap nodes. In an MIT proposal [DenGT84], an array may be represented as "the set of values conveyed at the same moment by a certain group of destination arcs", and, in the Manchester machine, as the elements on an arc, with the index field representing the array subscript [WatG82]. Such low level, machine dependent descriptions of data are utilized to gain greater efficiency from early data flow hardware. Higher level views of data structures may become as efficient in the future [CalDP83, ArvI83].

3.4.3 Using graphs with a data flow machine

The translators, described in Chapters 4 and 5, generate graphs which can be executed by the data flow graph interpreter. It is anticipated that these translation techniques would be applicable to the generation of code executable on specific data flow machines. This section looks at extensions necessary to extend the techniques in this way.

Two approaches are possible, either refinement of the code generator, or transformation of the graphs produced by the translator. As the latter is the more general method, an assessment is now made of the extent to which graphs executable by the interpreter correspond to programs executable on data flow machines.

Firstly, two of the more straightforward aspects of the transformation process are considered. The arithmetic instructions of the interpreter are generic in that they are applicable to both integer and real operands; it may be necessary to transform them into type-specific instructions, as in the MIT static machine [Tod81]. An IDENT instruction with multiple outputs would need to be converted to a series of instructions with at most two outputs for the Manchester machine [GurWG80].

Iterative schemes are commonly used in data flow machines [GurWG80, ArvGP78]. As shown in §3.4.1.1, corresponding schemes can be developed using the model of this chapter with only slight differences at the schematic level; it should be possible to transform graphs based on such schemes into machine-specific cyclic code with little difficulty. Function invocation and return should also be straightforward; for example, the standard apply/return interfaces [GurWG80] of the Manchester proposal could be used to replace the APPLY operation of the interpreter.

Structure operations may, however, require substantial transformation, for the reasons noted above. It would be interesting to explore the feasibility of developing standard transformations between common modes of use of structures and machine-specific representations.

3.4.4 Comments on early completion data structures

Early completion data structures are introduced to permit incremental creation of, and access to, data structures; this is achieved by separating the actions of creating, appending to, and selecting from, structures. However, when using these operations, some care must be taken to ensure that the structures created are indeed functional. For example, a structure may be created by a PAIR activity, and passed to two distinct activations, each of which execute a MKL activity on the structure. The first to be executed will succeed in replacing a queue with a value, and the second will fail because the component is not a queue; thus, the value once assigned will not be changed, but the value itself is determined by a non-deterministic "race" between the two MKL activities. In this thesis, the instructions are always used in a way which ensures functionality.

In §3.2, it was indicated that execution of a RETURN activity within an activation causes deletion of the activation from the heap. It should be noted that such a scheme is inadequate if the activation uses early completion data structures, because structure operations may be pending after the result has been returned. One solution is to require that the completion of such operations generate special signals which, together with a signal from the RETURN instruction, indicate that the activation can no longer generate activities.

It is noted in [Den81] that early completion data structures can be used to represent tuples of arguments to, and results from, functions. Consequently, function evaluation can commence with the arrival of any argument, and partial results can be returned as soon as they are generated. As this facility was not essential for the aims of the research, it was not implemented in the interpreter (see §7.4.1 for further discussion).

Other schemes, which do not depend on early completion data structures, have been developed to permit incremental passing of argument values to a function invocation. For example, the code template for calling a function suggested by Gurd et al in [GurGK81] uses a trigger to initiate the the steps required to set up a new activation, independently of the arrival of arguments. Caluwaerts et al [CalDP83, CalDP82] use a similar scheme. Amamiya et al [AmaHM82] pass incoming arguments through a network of specially designed or-gates which detects the arrival of the first argument of a new function invocation, triggering the creation of a new activation.

Notions similar to early completion data structures appear in other proposals. The I-structures of Arvind et al [Arv80, ArvG82], and the scheme of Amamiya et al [AmaHM82], were mentioned in §1.6.1. The Manchester machine provides a facility for deferred access to an array element [WatG82]; a special matching function, termed "preserve-defer", preserves a complete array at a single node, and permits accesses to be attempted before the array is formed; it does this by attempting the access repeatedly, rather than by queueing the read request. Kishi et al [KisYK83] use a wait queue to hold attempted read operations.

CHAPTER 4

THE IMPLEMENTATION OF LX3

4.1 Introduction

The traditional concept of assignment is not supported in the data flow model. However, in implementing a high level language, the left hand side of an assignment statement can be regarded as associating a name with an arc of a data flow graph, and the right hand side as defining values which flow on that arc. Such a statement is seen as definition of the value associated with a variable, rather than assignment to a storage location; this notion has resulted in the development of single assignment data flow languages [ArvGP78, ComHS80, GurGK81], in which a value is assigned to a variable in only one place in a program.

LX3 is a language which permits only one definition of each variable, and hence can be considered a single assignment language. Many proposals for high level data flow languages permit programs to be built using iterative constructs, conditional definitions and functions; it has been shown in §2.3.2.2 that an LX3 program can be understood in these terms. LX3 must, therefore, be regarded as a potential data flow language.

In this chapter, an implementation is described which analyzes an LX3 program to determine its constituent loops, and then constructs corresponding data flow graphs. A data flow scheme corresponding to each LX3 construct is presented, the structure of the implementation and the operation of its principal components are discussed, and details are given of the techniques used in compiling code from the schemes. Finally, LX3 is compared with other languages specifically designed for use with data flow machines.

The results of the loop analysis of an LX3 program can also be used to generate code for a sequential machine. Such an implementation is described, and compared with the data flow implementation. The implementation is then compared with other

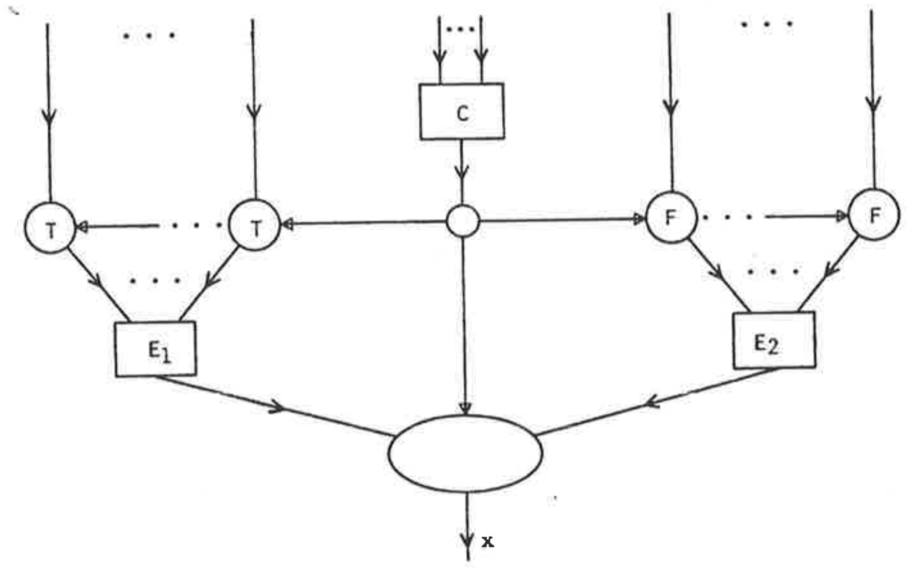


Figure 4.1. A data flow scheme for a conditional definition.

compiler based implementations of Lucid subsets.

An example of program translation appears in Appendix 3.

4.2 Code schemes for LX3 constructs

In this section, it is shown how each LX3 construct can be expressed in the graphical data flow language; these schemes provide the basis for compiling LX3. Later, the implementation of the transformations discussed here is described in detail.

4.2.1 Arithmetic expressions

The generation of code for arithmetic expressions is expressed as actions associated with the recursive descent analysis of the expression into terms and factors. Hence, the schemes used are not significantly different from those in many conventional compilers, and are not discussed further.

4.2.2 Conditionals

Figure 4.1 shows a data flow scheme for a definition of the form

$$x \;=\; \textbf{if } C \textbf{ then } E_1 \textbf{ else } E_2 \textbf{ eif}$$

assuming that the expressions E_1, E_2, and C are already compiled.

C, E_1 and E_2 are represented as boxes, with one input arc for each variable used inside the network represented by the box. The result of C is used as control input to TGATE instructions for each value used by E_1, and to FGATE instructions for each value used by E_2; hence, when the graph is executed, one, and only one, of E_1 and E_2 is executed.

4.2.3 Loops

The loop schemes used in this thesis are cyclic, for two reasons, firstly, such schemes have commonly been expressed in cyclic form in the data flow literature [Den74, Ada71, GurWG80, AllO79], and secondly to ensure that the schemes developed are usable with data flow machines based on cyclic models (see §3.4.1.1). The data flow model implements a loop as an acyclic tail recursive function template; details will be given in §4.4.2, of the transformation of the cyclic schemes into tail recursive form.

The essential requirement is a scheme for the circulation of values. For example, the cyclic graph of Figure 4.2, termed a simple circulator, permits the indefinite circulation of a single value, admitted to the scheme via the MERGE gate, and circulated while *false* values arrive at the gates shown. The MERGE gate used is the 2-input instruction described in §3.2.

It is usually required that the value of a variable for a new iteration be computed from that of the current iteration; this computation can be expressed as a data flow subgraph inserted at the point labelled 1 in Figure 4.2. The resultant scheme, shown in Figure 4.3, is called a circulator. A circulator is used to implement the computation corresponding to the definition of an inductive variable in LX3. Henceforth, A is used to indicate the circulator corresponding to the variable a, which, it will be recalled, has history $[\![a]\!]$.

Circulators may be linked together, as illustrated by the example in Figure 4.4;

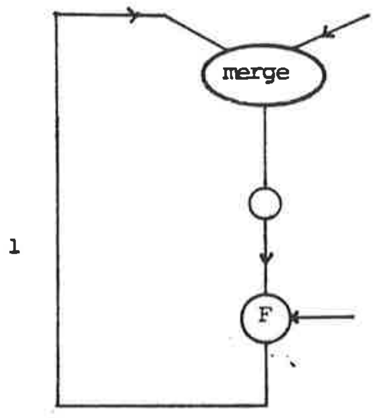


Figure 4.2. A simple circulator

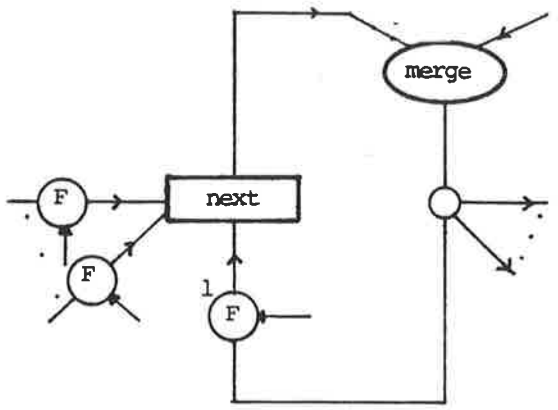


Figure 4.3. A circulator.

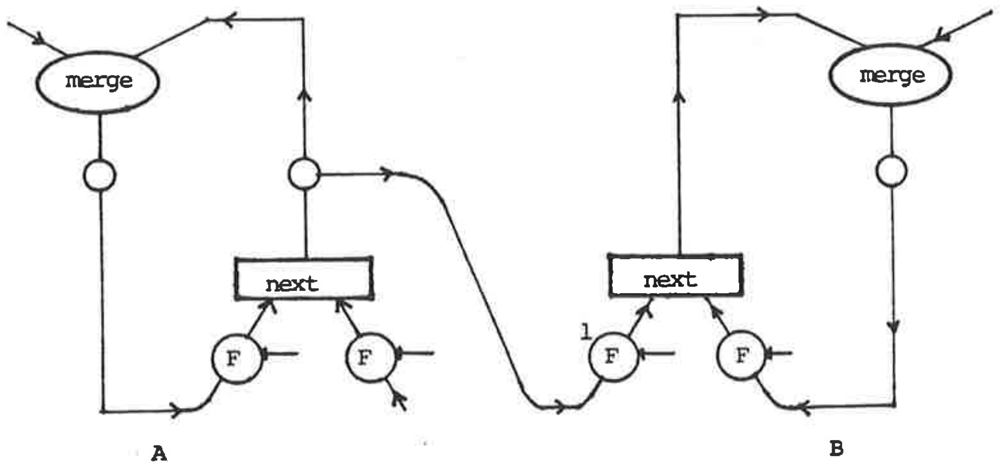


Figure 4.4. Interconnected circulators.

here, the FGATEs of each circulator receive the same sequence of control values, and a value from circulator A is used to compute an updated value in B. The dependency of *b* upon *a*, and hence of B upon A, implies the existence of a cyclic scheme in which A and B are components being driven by the same control values, and hence can be regarded as iterating together. An LX3 loop can be implemented from interconnected circulators, simple circulators and a subgraph representing the termination condition of the loop; the latter produces a sequence of control values common to all gates of the loop. The result of the loop is computed from values produced by the final iteration.

A generally applicable loop scheme must return to its initial configuration on termination of a particular loop execution; schemes with this property are said to be serially reusable [Rei78]. On termination, the FGATE instructions in Figures 4.2, 4.3 and 4.4 absorb superfluous values present in the subgraph and so ensure serial reusability. It should be noted that, once transformed into an acyclic form suitable for the data flow graph interpreter of Chapter 3, it is not necessary for loop schemes to be serially reusable, because each loop iteration is a separate activation, which ceases to exist once a value is returned from that iteration; any superfluous values also disappear. A similar effect has been noted by Treleaven et al [TreHR82]. However, schemes used in the implementation of LX3 are serially reusable, to ensure that all results derived are also applicable to cyclic schemes in which the property is important.

In LX3, loops involve three categories of variables, inductive, auxiliary and quiescent (see §2.3.2.2). An inductive variable is updated on each loop iteration and hence requires a circulator. An auxiliary variable usually defines an intermediate result, expressed in terms of values available within a single iteration, and its value need not be circulated. Some quiescent variables are used in each iteration of a loop; a simple circulator must be generated for each such variable to ensure that its value will be available at each iteration.

Use of the circulator of Figure 4.3 is based on the assumption that the definition of the **next** attribute of an inductive variable does not use the **next** attribute of any other inductive variable. Consideration is now given to the generation of circulators when this restriction is lifted.

First consider the case when the value of an inductive variable depends on the **next** value of another inductive variable of the loop. The example of Figure 4.4 is used to explain how the definition

$$b \;=\; 1\ \mathbf{fby}\ b + \mathbf{next}\ a$$

is translated to a data flow subgraph. Circulators A and B correspond to the two inductive variables, a and b. Suppose that C denotes the termination condition of the loop, as determined by the appropriate **asa** definition, and that the loop is terminated at some iteration N such that control value C_N is *false* and C_{N+1} is *true*. Then a_N (that is, $[\![a]\!]_N$) and b_N are the final "current" values produced by A and B respectively, and a_{N+1} is the value of **next** a when the loop terminates. It can be seen from Figure 4.4 that the arrival of C_i permits the transmission, through the FGATEs of B, of b_{i-1} and a_i, the values required for the computation of b_i. Hence, the arrival of C_{N+1} should absorb b_N and a_{N+1}; however, the FGATEs of A inhibit the production of a_{N+1}, which, therefore, does not arrive at the FGATE labelled 1, so a superfluous control value is left in the circulator B. Hence, in this situation, the FGATE 1 is not required, as the value it is designed to absorb is never produced; in fact, the gate must not be present if the scheme is to be serially re-usable.

A different arrangement of gates is required if a_{N+1} is actually used, as, for example, in the definition

$$x \;=\; \mathbf{asa} \ldots \mathbf{then}\ 5{*}\mathbf{next}\ a\ \mathbf{easa} \qquad \ldots(2)$$

where the value a_{N+1} is needed to compute the result of the loop. In Figure 4.4, the FGATEs of A are used to suppress the computation of a_{N+1}, and, if they are removed, the computation will proceed. However, FGATEs are then needed at 1, to absorb a_{N+1}, and immediately preceding the MERGE gate of A, to prevent a superfluous value of a_{N+1} reaching the MERGE gate.

If b_{N+1} is needed in a computation, then a_{N+1} is also needed, because b depends on a. In this case, a gating arrangement similar to that discussed in the preceding paragraph is required for both A and B, except that the FGATE 1 is not required, as a_{N+1} must be computed.

Thus it is necessary to analyze each use of the **next** value of an inductive variable a and, in particular, determine whether or not a_{N+1} is required. With the results of this analysis available for all the inductive variables of a loop, it is possible to formulate the following rules for the generation of gates in a manner which ensures serial re-usability of the data flow graph.

Associated with each inductive variable i, is a predicate *nplus*(i), defined to be *true* if the value i_{N+1} must be computed. For example, in definition (2) above, *nplus*(a) holds because, when the loop terminates, the value of **next** a, namely a_{N+1}, is required to compute x. In general, *nplus*(i) is true if either of the following conditions hold:

- There is a quiescent variable x such that x is dependent, directly or indirectly, on **next** i
- There is an inductive variable j such that *nplus*(j) holds and **next** j is dependent, directly or indirectly, on **next** i, and hence i_{N+1} is needed to compute j_{N+1}.

The following four gating rules can now be stated. The last three rules refer to two inductive variables a and b such that **next** b depends on **next** a.

(1) If nplus(i) holds, then an FGATE is needed immediately preceding the MERGE gate in the circulator for i.

(2) If (*nplus*(a) $\wedge$ *nplus*(b)) holds, then no FGATEs are needed on the **next** network of either a or b.

(3) If *nplus*(a) is *false*, then *nplus*(b) must be *false*, and FGATEs are needed for the **next** networks of both a and b, except that the uses of a implied by the dependency of **next** b on **next** a must not be gated.

(4) If (*nplus*(a) $\wedge \neg$ *nplus*(b)) holds, then no gates are needed for a, and similarly for b, except that uses of a must be gated.

4.2.4 Define clauses

The interface between an unparameterized **define** clause and the network which uses it, is defined by the data flow code of the clause, its set of global variables, and its *result* variable. In this implementation, a function template is generated from the clause, and linked into the enclosing network as a nested loop, using the loop activation instruction IAPPLY.

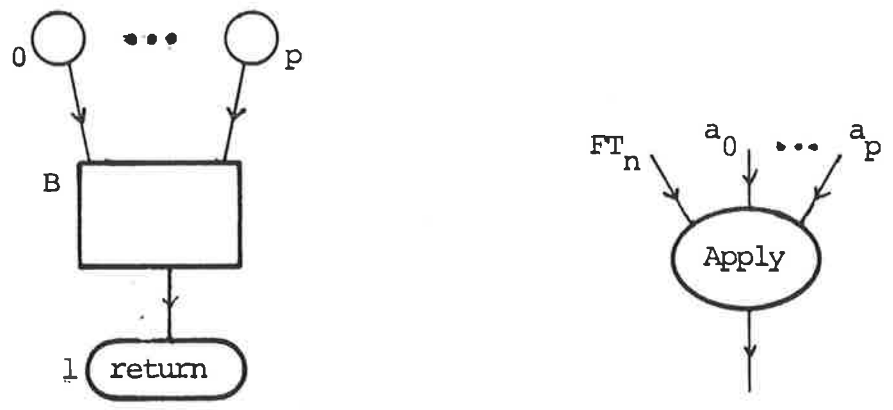


Figure 4.5. Data flow schemes for a **define** clause and its invocation.

A parameterized **define** clause is similar to a **mapping** definition [AshW78], and is compiled as a data flow function template. A use of such a clause is regarded as an invocation of that template. Figure 4.5a illustrates the scheme used in compiling a parameterized **define** clause; the box labelled B represents the data flow code generated from the definitions of the clause, and the IDENT instructions the interface for passing parameter values. Figure 4.5b shows an invocation of the clause.

4.3 Implementation of LX3

4.3.1 Structure of the implementation

The implementation is organized as shown in Figure 4.6. Syntax analysis is performed using recursive descent techniques; within that framework, there are procedures for identifier table maintenance, and for generation of primitive code. The source analyzer generates a function template for the main program and for each **define** clause; it also produces an incomplete data flow subgraph for each LX3 definition, but it does not have sufficient information about loops to complete the generation of loop code, a consequence of the lack of ordering of definitions in an LX3 program. The source analyzer cannot determine either the structure of loops or which constants need to be circulated in a loop; instead, it produces a dependency graph.

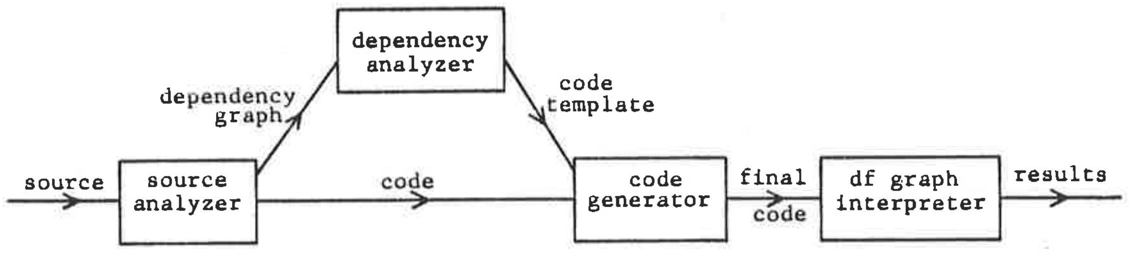


Figure 4.6. Implementation organization.

The dependency analyzer traverses this dependency graph to determine the loop structure of the program, and to determine the gating appropriate to individual circulators within each loop. The results of this analysis are recorded on the code template (described in the next section).

The code generator scans the code template and makes any necessary alterations to the original data flow code produced by the source analyzer. Finally, the code generator also determines from the resultant data flow code where triggers are required, and generates triggering arcs for CONSTANT instructions from preceding instructions.

As indicated above, the language LX3 is such that a compiler must perform some loop analysis before code generation can be completed. The dependency analyzer is responsible principally for the analysis of loops, while the source analyzer and code generator cooperate in performing the generation of code. In the sequel, the principal data structures of the implementation are first described, and then the implementation of code generation for each of the above schemes is presented in detail. As the dependency analyzer determines many of the actions required of the code generator, it is convenient to discuss the implementation of these actions during the description

```
type
  idtablecell =
    record
      ident: alfa;
      level: levrange;
      charaddr: coderange;
      deptr: ↑deplist;
      nplus: boolean;
      kind: (aux, quiescent, inductive);
      gatelist: ↑deplist;
    end;
  deplist =
    record
      arc:
        record
          suc: idtableaddr;
          attr1, attr2: 0..2;
        end;
      occ: ↑occlist;
      next: ↑deplist
    end;
  occlist =
    record
      lk: link;
      next: ↑occlist
    end;
  link =
    record
      instr: instrno;
      posn: prange
    end;
```

Figure 4.7. Identifier table structure.

of the dependency analyzer.

4.3.2 Principal data structures

The identifier table, with an associated dependency graph, and the code template, are the principal data structures of the implementation. Pascal type declarations for the identifier table and associated types are given in Figure 4.7; those type declarations left unspecified are declared as suitable integer subranges.

Consider the type *idtablecell* shown in Figure 4.7. The *ident* field contains the identifier. A level number is associated with each clause; the main program is at level 1, and the level is incremented for each inner clause. The field *charaddr* indicates the first of the three contiguous characteristic addresses associated with the three attributes of the identifier. This field, which is allocated on creation of the table entry for the identifier, holds the address of a data flow instruction from which values of the attribute can be taken when required during compilation.

The dependency graph records dependencies between attributes of variables. Each identifier appears as a node of the graph. A directed arc from node *a* to node *b* indicates that an attribute of *a* depends on an attribute of *b*, the nature of the dependency being shown by labelling the arc with an ordered pair of digits to represent the attributes of its initial and final nodes; the **first** attribute is numbered 0, the "current", 1, and the **next**, 2.

For an identifier *a*, the field *deptr* of its identifier table entry is the pointer to its dependency list. A dependency list element represents an arc of the dependency graph (see Figure 4.7), and records the fact that a particular attribute of some identifier is used in the definition of *a*; such a use may occur once or more in the definition, so a sublist, the *occlist*, associated with each dependency list entry, records each point of use in the data flow code. Both the dependéncy list and the *occlist* are updated as the source code definition is parsed from left to right.

The flag *nplus* implements the predicate *nplus*(*i*) defined in §4.2.3; the field *kind* is self-explanatory. The *gatelist* is a copy of part of the dependency list for the identifier; it contains the dependency list entries pertaining to the definition of the **next** attribute of an inductive variable.

The code template is the interface by which the source and dependency analyzers communicate with the code generator. The information contained in each entry is largely inherited from the appropriate identifier table entries, namely *ident*, attributes, characteristic addresses, *occlist*, *gatelist* and flag *nplus*. The template is structured according to the level and loop structure of the program; markers separate the section for one clause from that for the next, and, within each such section, additional markers separate the contributions made by each loop of the clause. An example of a code template is given in Appendix 3.

Code generation is expressed in terms of a record description of an instruction, an array of instructions containing the data flow code, and operations (implemented as procedures) on this array. Operations are provided to generate an instruction, to create an arc between two instructions, and to find the next available position for the extraction of a value from a characteristic address.

4.3.3 The dependency analyzer

The primary function of the dependency analyzer is to determine the definitions which constitute a given loop; the **asa** definition associated with each loop provides the basis for doing this. As all identifiers involved in a loop must contribute to the result defined by the **asa** definition, the variable x on the left hand side of the definition is dependent, either directly or indirectly, on each loop identifier. It follows that all identifiers in the loop can be found by examining the substructure of the dependency graph starting at x. Whenever this traversal encounters an **asa** definition, loop analysis is invoked and continues while the graph traversal covers the substructure of x. Lists of quiescent and inductive variables are accumulated during the traversal; the concatenation of such lists from every loop of the program produces the code template. The operation of the dependency analyzer is now described in more detail.

Note that, in an LX3 **define** clause, a use of a non-local variable does not imply a dependency on that variable, because its value will be constant within the clause. Thus, the dependency structure of each clause can be analyzed independently of the rest of the program. The overall structure of the dependency analyser is as follows:

```
template <- nil;                { the code template }
for each clause of the LX3 program do
   fixloop( r ).
```

r is the identifier table entry of the variable defined as the result of the main loop of the clause. In the case of the program clause, r will not necessarily be the variable *result*; it is the variable defined with an **asa** definition which is not used in any other definition, for example, *stop* in Figure 2.7. The definition of the procedure *fixloop* is given in Figure 4.8, in which the template entry *lv* contains information about the termination condition and result of the loop.

```
procedure fixloop( lv )
   { lv is an identifier table entry representing }
   { a variable defined by an asa definition. }
   { The operator + is list concatenation }
   var q, i  : v_list          { lists of quiescent and inductive variables }
   begin
      q <- nil; i  <- nil;
      searchtree( lv );          { builds q and i }
      template <- template + lv + q  + i;
      fixgates( i );             { see text }
   end
```

Figure 4.8. Definition of procedure *fixloop*.

The procedure *searchtree*, a local procedure of *fixloop*, is defined in Figure 4.9. It is used to perform a breadth-first search of part of the dependency graph.

```
procedure searchtree( st )
   { st is an identifier table entry, representing }
   { the root of substructure to be searched }
   begin
      if st is defined by an asa definition
      then if  st <> lv  then  fixloop( st )
      else
         begin
            for each node, nd, in the dependency list of st do
                  processnode( nd );
            for each node, nd, in the dependency list of st do
                  searchtree( nd );
         end
   end
```

Figure 4.9. Definition of procedure *searchtree*.

The procedure *processnode*, local to *searchtree*, is defined in Figure 4.10. The generation of a simple circulator is required to circulate the value of a quiescent variable only if it is used by each iteration, specifically, in the definition of an auxiliary variable, or in defining the **next** attribute of an inductive variable; if it need not be circulated, the code generated by the source analyzer is adequate, and no action is required. Hence, quiescent variables which require further action by the code generator are recorded on the code template.

```
procedure processnode( node )
  { lists q and i are inherited from fixloop }
  begin
    if node corresponds to a quiescent variable
    then
        if value must be circulated in loop
        then record node on list q
        else no action required
    else if node corresponds to an inductive variable
    then
        if dependency involves  next  attribute of node
        then
          begin
            determine setting of node.nplus;        { see text }
            record node and nplus on list i
          end
        else
            same action as for a quiescent variable
  end
```

Figure 4.10. Definition of procedure *processnode*.

A second important function of the dependency analyzer is to implement the gating rules given in a previous section. The dependency analyzer evaluates the predicate *nplus*(*i*) for each inductive variable *i*, and, when analysis of a loop is otherwise complete, uses the dependency relations between each pair of inductive variables to determine gating requirements according to rules (1) to (4) above, and record the predicate value in the code template. The code generator, as it scans the code template, can then determine whether or not to generate a gate for a particular use of an inductive variable. A more detailed description of the techniques used is now given.

Within the dependency analyzer, the flag *nplus* associated with each identifier table entry is set. As loop analysis proceeds, the processing of inductive variables includes the necessary checks described in §4.2.3 to determine the value of *nplus*. In the current implementation, these checks are incomplete, in that only direct dependencies are checked; indirect dependencies did not arise in the examples tested, but their inclusion would require only a list of identifier uses for each identifier table entry, and some modification of the dependency analyzer. The value of *nplus* is recorded on the code template element associated with the inductive variable; subsequently in the code generator, if *nplus* holds, a gate is generated between the **next** network and the MERGE instruction of the associated circulator, and no other gates are generated (Rules (1), (2) and (4)); otherwise further checks must be carried out, as described below.

The code template element for an inductive variable ***i*** includes the *gatelist* for ***i***, which, as explained in §4.3.2, records that portion of the dependency list relevant to the computation of **next *i***. Also associated with each *gatelist* element in the template is a flag which indicates whether or not a gate must be generated for the kind of use specified. The code generator checks this flag as it scans the *gatelist*, and, if necessary, generates a gate at each point of use specified in the associated *occlist*.

The procedures provided in the dependency analyzer for the implementation of the rules of §4.2.3 are now discussed. As mentioned above, determination of *nplus* takes place in the procedure *processnode* during dependency graph traversal. Rules (2) to (4) are implemented primarily by the two procedures *fixgates* and *fixarcs*, the definitions of which are shown, respectively, in Figures 4.11 and 4.12. It can be seen from Figure 4.11 that the default setting of the gate flags is changed by *fixarcs*, if necessary. Procedure *fixarcs* is used to check for uses by inductive variable *iv* of other inductive variables, a condition which causes one of Rules (3) and (4) to become relevant.

```
procedure fixgates( il )
  { il is a list of inductive variables }
  begin
    for each inductive variable, iv, of il do
      begin
        if iv.nplus
        then set iv.gatelist.flags for no generation of gates
        else
          begin
            set flags for generation of all gates;
            fixarcs( iv.gatelist )
          end
      end
  end
```

Figure 4.11. Definition of procedure *fixgates*.

4.3.4 Checking the subset restrictions

The definition of LX3 §2.3.2.1) states seven restrictions on LX. Restrictions (1)

```
procedure fixarcs( g )
  { g is a gatelist }
  begin
    for each element, y, of g
      begin
        if the variable identified by y is inductive
        then
            if y.nplus
            then set flags appropriate to Rule (4)
            else set flags appropriate to Rule (3)
      end
  end
```

Figure 4.12. Definition of procedure *fixarcs.*

to (3), and Restriction (7), are entirely syntactic, and can be checked accordingly. Restriction (4), requiring the presence of an **asa** definition of *result*, can be checked at the end of syntax analysis of a clause.

Dependency analysis can be used to check Restriction (5), which requires that no identifier be defined in terms of its own future. In LX3, the only permitted access beyond the current iteration is via the **next** attribute of an inductive variable; the restriction is violated if the current attribute of a variable x depends, directly or indirectly, on the **next** attribute of x. As dependencies between attributes are recorded on the dependency graph, it is straightforward for the dependency analyser to examine such dependency chains. Circular dependencies can be checked similarly.

Restriction (6) is checked by examining each variable x in the expression defining the first operand of **fby**; as clause subjects are not quiescent, they are excluded by this restriction from such expressions. Each variable in the substructure of the dependency graph starting at x must be either qualified by **first** or defined by an **asa** expression, or be dependent directly or indirectly, on only such variables. This restriction can be relaxed by extending the implementation, but this was not found necessary for the investigations reported in the thesis.

4.4 Generation of data flow code

4.4.1 Conditionals

It can be seen from the conditional scheme of Figure 4.1 that each input to E_1 or E_2 must be gated, and that in providing these gates the set of input variables to each of E_1 and E_2, and their points of use, must be determined. Sufficient information is recorded on the dependency list and its associated *occlist* to permit these gates to be generated by the source analyzer.

In finding the sets of input variables and their uses, it is necessary to determine which occurrences were added during compilation of C, E_1 and E_2. If I_{E_1} is the set of input variables to E_1, and O_{E_1} the variables in *occlist* after E_1 has been parsed, then

$$\begin{aligned}\mathrm{I}_{E_1} &= \mathrm{O}_{E_1} - \mathrm{O}_C \\ \mathrm{I}_{E_2} &= \mathrm{O}_{E_2} - \mathrm{O}_{E_1} - \mathrm{O}_C.\end{aligned}$$

The fact that the dependency list is built during a left to right scan of the definition simplifies the computation of these sets.

In providing gates for input variables, it is simplest to gate each use of a variable, but this causes an unnecessarily large number of gates to be generated. Hence, a more economical method is used, generating one gate for each input variable, and using an IDENT instruction to distribute the gated value to each point of use.

4.4.2 Loops

As discussed above, the source analyzer generates circulators and other data flow code for the individual definitions of an LX3 loop, and the dependency analyzer determines the overall structure and gating requirements of the loop, recording this information on the code template.

An inductive variable *i* is defined using the **fby** construction. The source analyzer generates part of the code required by the circulator scheme of Figure 4.3, namely the code for the expressions defining the attributes, the MERGE gate, and links to the data inputs of the MERGE gate. The *gatelist* is also constructed. When an **asa** definition is encountered, the source analyzer generates code for the expressions defining the condition and result specified. All gates and control links are generated by the code generator, as described previously.

The transformation of such a loop into the tail recursive form required by the data flow model is now discussed. As discussed in §3.2, a single instruction, IAPPLY, is used to admit the initial values of all circulators into the loop; the loop itself is a separate FT, activated by the execution of an IAPPLY instruction with the initial values as parameters. A new iteration is set up, not by linking the **next** network to the MERGE gate to form a cycle, but by sending the updated value of each circulator to a single INCR instruction, which creates a new activation for the next iteration, with the updated values as parameters. A RETURN instruction is used to return the result of the loop.

To effect the transformations required, the cyclic loop scheme is modified as follows. IAPPLY, INCR and RETURN instructions are generated. The output from the **first** network of each circulator is linked to an input of the IAPPLY instruction. The IAPPLY instruction, when executed, passes its parameter values into the FT in the same way as an APPLY instruction; hence, within the FT, an instruction which delivers the appropriate parameter value replaces the **first** network. Similarly, the arcs which transmit updated values to the separate MERGE instructions of each circulator are removed, making the graph acyclic, and connected, instead, to the appropriate input of the INCR instruction; transmission of these updated values to the next iteration is accomplished by the parameter passing mechanism. Finally, the instruction which delivers the result of the loop is linked to the RETURN instruction.

Note that, subsequently in the thesis, the term "circulator" is used in reference to both the cyclic and equivalent acyclic schemes.

4.4.3 Define clauses

A **define** clause introduces a new level of lexical scoping in an LX3 program. Hence, a new level is allocated in the identifier table, and the local variables of the definition included at that level. The list of definitions which makes up the body of the clause is compiled using the techniques discussed so far.

Each variable included in the **using** list of the clause is quiescent inside the clause, as its value is frozen within the inner loop. This effect is achieved by adding to the identifier table for the clause, a new entry for each such variable, and marking it as quiescent. Each such entry has a new characteristic address, from which the input value of the variable is sent to each point of use within the network for the clause. The appropriate values are transmitted to the clause, and thence to the new characteristic addresses, when the data flow graph representing the clause is invoked.

Clause subjects on the **using** list are treated slightly differently. Consider an inherited subject *f*. The frozen values of its globals must be inherited implicitly, as explained in §2.2.2.2.5. This is implemented by replacing the identifier *f* on the **using** list with a list of its global identifiers; the process of replacement continues until no subject identifiers remain on the **using** list.

It will be recalled, from §2.3.2.2, that an unparameterized LX3 **define** clause is viewed operationally as a nested loop, whereas a clause with parameters is regarded as akin to a function. Consequently, in implementing clause invocation, the two cases are treated differently.

For an unparameterized clause, the input interface is formed by linking the old characteristic address in the enclosing network to an IAPPLY instruction which activates the nested loop, and thence to the new characteristic address, making it possible to pass input values to the network for the clause. The result interface is formed by linking the IAPPLY instruction to the characteristic address of the variable being defined, so that the result of the nested loop is passed back to the enclosing network. Note that direct linking of a subgraph in this fashion is shown to greater advantage in a cyclic scheme, where it permits a nested loop to be used without the overheads of function application [ArvGP78, GurWG80].

For a parameterized **define** clause, an APPLY instruction is generated each time the clause is used in the text of a definition. Input links for parameter values and values of variables on the **using** list are linked directly to the APPLY instruction. The output of this instruction is directed to the point of use of the clause value.

4.5 A Sequential Implementation of LX3

The implementation structure described above can also be the basis for the translation of LX3 into a sequential target language. A description of such an implementation is now given, with emphasis on the differences from that described above.

4.5.1 Target language

To avoid undue emphasis on the details of code generation, it was considered desirable to choose a simple target language. It was thus convenient to use the instruction set of the PL/0 machine, described by Wirth [Wir76], as the target language. This instruction set is simple, sufficiently powerful for the purposes of this experiment, and a suitable interpreter was readily available at the time.

The PL/0 machine has a stack oriented architecture. Its instruction set includes instructions for transferring words between top of stack and memory, instructions for stack maintenance, jump instructions and arithmetic and relational operations.

4.5.2 Structure of the implementation

The source analyzer builds a dependency graph, as described above. It also produces a piece of code for each definition in the program; these pieces of code produced by the source analyzer are unordered.

A major function of the dependency analyzer, in addition to loop analysis, is to determine a suitable order for the execution of code. The code template is used to indicate where each of the previously generated pieces of code is to appear, and where code needs to be added to ensure correct execution of loops.

The code generator uses the code template to rearrange the pieces of code generated by the compiler in an appropriate order, with some additional code for loop control. The result is ordered code, in a form suitable for execution on the PL/0 interpreter.

4.5.3 Principal data structures

The principal data structures used are, as before, the identifier table, incorporating a dependency graph, and the code template. Differences are now explained.

The characteristic address field of the identifier table is interpreted as the address of a group of three locations used to store the values of the **first**, **next** and "current" attributes of an inductive variable; for other variables, only the "current" attribute is defined. The code address field stores the address of the piece of code generated from the definition associated with the identifier; this piece of code may be re-positioned by the code generator. The flag ***nplus***, and the sublists associated with each dependency list entry in the data flow implementation, namely the ***gatelist*** and the ***occlist***, are not used in this implementation.

4.5.4 Dependency analyzer and code generator

The dependency analyzer must determine the order in which code for definitions is evaluated. The analysis of the dependency graph is based on a tree traversal algorithm, modified to allow for nodes which have more than one incident arc; nodes are marked when first traversed, and further traversals through such nodes are not permitted. A postorder traversal, which visits first the left subtree, then the right subtree, and then the root node, gives an appropriate ordering for the execution of pieces of code.

The loop analysis function of the dependency analyzer proceeds as described previously. During this phase, ordered lists of inductive, auxiliary and quiescent variables

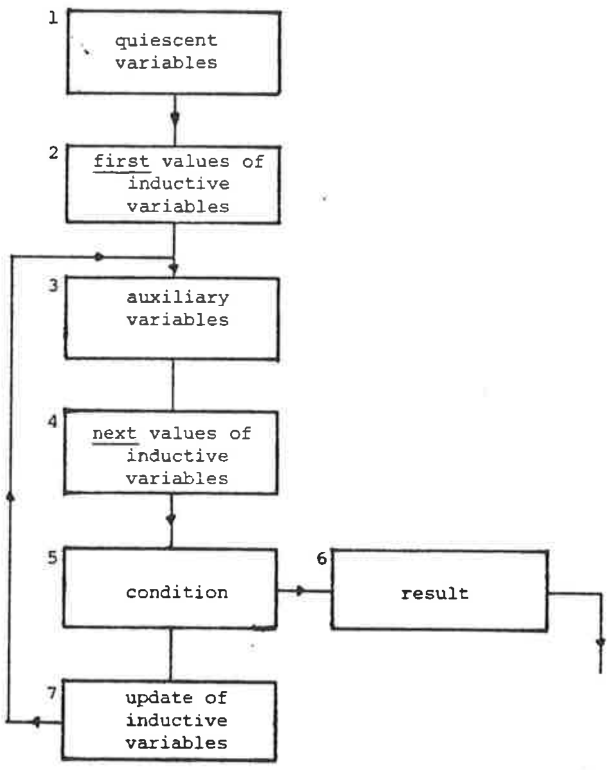

1
quiescent variables
2
first values of inductive variables
3
auxiliary variables
4
next values of inductive variables
5
condition
6
result
7
update of inductive variables

Figure 4.13. A scheme for sequential code.

are accumulated, and on completion of the loop analysis, these lists are merged into a single list following the pattern of the loop scheme, shown in Figure 4.13. This list shows which variables appear in each section of the loop scheme, and also contains markers which indicate where boundaries between sections occur. Later, branching code is inserted at some of the points where markers occur. Application of this analysis to each loop eventually produces a single list, in which every variable in the original dependency graph has been ordered. Such lists exist for every clause in the program, and when concatenated, constitute the code template.

The function of the code generator is to use both the control template and the pieces of code produced previously, to generate code which is directly executable. For every clause, this is carried out by traversing the associated list of variables, the list being part of the code template; the appropriate piece of code is relocated if the list element represents a variable; some loop branching code is generated if the list element is a marker. Thus, each loop defined by the LX3 program is translated into a target language iteration, patterned after the loop scheme of Figure 4.13.

4.6 Discussion

4.6.1 Comparison with data flow languages

Several high level language proposals have been presented for data flow machine architectures based on the Dennis data flow model [Den74], principally at MIT (VAL [AckD79]), the University of California at Irvine (Id [ArvGP78]), the University of Manchester (Lapse [GurGK81]), and Iowa State University (a Pascal-like language [AllO79]). The languages VAL, Id and Lapse are value oriented, single assignment languages, each of which includes constructs for binding of an identifier to a value, for conditional definition, and for iteration. In a single assignment language, an identifier can have a value bound to it just once in its scope; this property is natural in a data flow language because an association can be made between an identifier and an arc of a data flow graph. LX3 also is a single assignment language, as each identifier is defined once only.

LX3 has been defined by imposing restrictions on tha language LX (see §2.3.2.1). These restrictions are now considered in relation to other data flow languages; it will be shown that LX3 is a language comparable in expressiveness to other data flow languages. Restriction (1) requires that **first** and **next** be considered as attributes of an inductive variable. This is very similar to the treatment of updated loop variables in other languages; in fact, LX3 goes further than some in permitting general use of the **next** attribute. Freezing of globals and parameters, as required by Restriction (2), is generally accepted in other languages; the restriction excludes those features of LX which extend the semantics beyond a conventional interpretation, as does Restriction (3) in omitting nonpointwise operators

The requirement expressed in Restriction (4), that the body of a **define** clause be a loop, is somewhat unusual, and encourages some artificial constructions in programming. However, it is not essential to the definition of the language, and could be relaxed by using more sophisticated dependency analysis; such extensions are beyond the scope of this thesis.

Restriction (5) again excludes certain features of LX which do not have a conventional interpretation. There are no operators in the other data flow languages considered so far which permit the current iteration to depend on future iterations.

Restriction (6) effectively prevents initial values of a loop being extracted directly from inside an iteration. Such a facility is not usually available in other languages; in general, a loop would have to return such values as extra results of the loop. Note, however, that any value can be inherited by a **define** clause and used as the initial value of a nested loop.

This Chapter emphasizes implementation of the basic constructs of LX3, and does not consider the implementation of structures, which are omitted from the language (Restriction (7)). Such features are essential if the language is to be developed further; for example, an array structure, with appropriate aggregate operators analogous to the **forall** construct of VAL [AckD79], would be desirable.

It can be concluded that LX3 is a suitable candidate for use as a data flow language, in that it can express the basic constructions of definition, conditional, iteration and function invocation. It has been shown that each of these can be implemented efficiently, using essentially standard translation techniques.

It has been suggested that conventional languages can be used, in a modified form, as data flow languages [AllO79, Vee81]. Allan and Oldehoeft [AllO79] propose a Pascal-like language without a **goto** statement or global references, and in which procedure parameters must specify a directionality (either in, out, or both). Because several assignments may be made to variables in the language, data flow analysis is used to determine the definitions and uses of a variable; a data flow graph arc can then be associated with each re-assignment to a variable. The translation of a conditional statement (as distinct from a conditional expression) requires further data flow analysis to determine the input and output sets of the statement; the corresponding data flow graph must have an input arc for each variable in the input set, and an output arc for each member of the output set. Similar data flow analysis is performed on iterative statements; a scheme like that of Figure 4.14 is then used in translation.

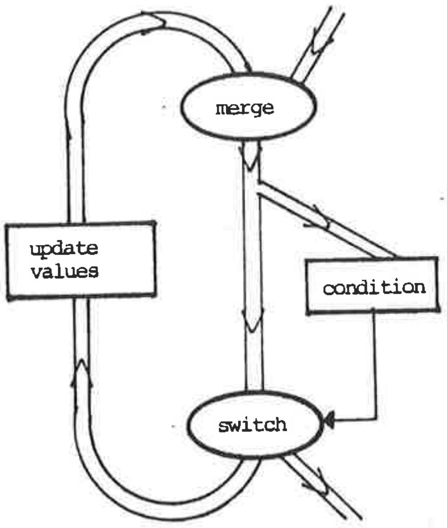


Figure 4.14. A general loop scheme.

Veen [Vee81] describes compilation techniques for a conventional language which includes global variables and procedures with side effects; if a global variable is used by a procedure it is regarded as an additional input parameter to the procedure, and if updated it is regarded as an extra result of the procedure. It is here that one of the difficulties of compiling a traditional language in a data flow environment is seen; for updates to global variables to be implemented correctly with this method, it is necessary to perform an exhaustive data flow analysis at compile time to determine all possible side effects of every procedure. Furthermore, sharing of global variables between procedures often requires sequential execution of procedure invocations which could otherwise be executed in parallel. This is but one illustration of the inherently sequential nature of conventional languages, and supports an argument for the use of languages which are fundamentally non-sequential, in which parallelism arises more naturally and is easily exploited. The work reported here confirms the view that

LX3 is indeed such a language, along with the single assignment languages discussed above, and applicative languages such as the language described by Friedman and Wise [FriW76], KRC [Tur81], FGL [KelLP78] and CAJOLE [HanG81].

4.6.2 Comments on loop schemes

The schemes described for arithmetic expressions and conditional definitions are essentially the same as those used in proposals for VAL, Id and Lapse. However, considerable differences between the implementation of these languages and that of LX3 emerge when schemes for iteration are considered. Each of the languages VAL, Id and Lapse has a syntactic unit for the expression of iteration, which, semantically, can be seen as accepting input values, of which some are used as initializations of variables updated in the loop and the remainder as values which remain constant for the duration of the loop.

Figure 4.14 shows a general data flow loop scheme which captures the essential features of the schemes used in other implementations [ArvGP78, GurGK81]. In this scheme, MERGE gates permit the introduction of initial values and the circulation of updated values, and SWITCH gates permit the circulation of values during the iteration, and, at the last iteration, absorb unused values and transmit result values.

The expression of iteration in LX3 differs in two ways. Firstly, there is no single syntactic unit corresponding to a loop. This need not necessarily affect the scheme used, but requires a method of synthesizing the loop from its individual definitions; this has been accomplished by using dependency analysis to find which definitions are needed to compute the result of a loop.

Secondly, both "current" and **next** attributes of an inductive variable can be used in definitions; in terms of the loop scheme, a value associated with the next iteration can be used both in the current iteration, and in computing the result of the loop. In VAL, either the old or the new value can be used, as the order of definition of updated values is significant, and a somewhat different loop scheme is used [BroM79].

The scheme of Figure 4.14 is inadequate if updated (**next**) values are returned as results of the loop. Attempts to use the scheme to return such values usually cause superfluous values to remain in the loop scheme at termination, thereby destroying the property of serial reusability. An analysis of the gating requirements for this case has been presented, rules for the generation of code developed, and a new serially reusable scheme presented.

In the new scheme, and in contrast to previously published schemes, gates are generated at the points of use of values, rather than at the origin of a value, as in Figure 4.14. By permitting greater flexibility in the placement of gates, this facilitates implementation of the gating rules of §4.2.3; for a typical circulator (Figure 4.3), these rules may require that any of the gates controlling input of values to the **next** network be omitted; even the gate admitting the current value of the circulator itself may be omitted.

4.6.3 Comparison of the data flow and sequential implementations

Some comments are now made about the differences between the individual phases of the data flow and sequential implementations. In the first phase, the source analyzer makes a single pass over the source program, producing code for each LX3 definition independently. In the case of compilation to a sequential target language, many unordered pieces of code are produced, but each piece of code, corresponding to one LX3 definition, is produced as it would be for a conventional high level language. In the case of a data flow target language, much of the dependency information inherent in the LX3 source program, is expressed directly in the data flow code itself, and further ordering by the translator is not required. In this sense, then, LX3 is a language which is implemented naturally on a data flow machine. However, loop analysis is required in both cases because loops are expressed indirectly in LX3 and no special ordering of the definitions associated with one loop is required. In the data flow implementation, the function of the dependency analyzer is only to determine the loop structure of the program, but in the case of sequential object code, the dependency analyzer is also used to establish the ordering of the pieces of code compiled from the definitions of the program; similarly, the final code generating phase is more complex for compilation to sequential code.

4.6.4 Other compiler based implementations of Lucid subsets

The compilation of Lucid programs has also been considered by Hoffmann [Hof78] and Farah [Far77]. Hoffmann has developed a system which enables Lucid programs to be compiled and executed, while Farah presents a theoretical scheme for the transformation of Lucid programs into an Algol-like language, and vice versa.

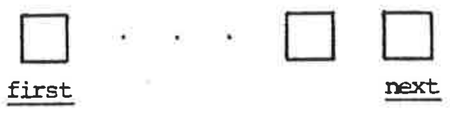


Figure 4.15. A window.

In LX3, **first** and **next** are regarded as attributes of an inductive variable, rather than operators on histories. In writing programs in LX3, a variable can still be regarded as a history consisting of a sequence of values, with the programmer being able to express relationships between variables in terms of **first**, "current" and **next** values. These three values can be regarded as a window through which the history of a variable may be viewed, as illustrated in Figure 4.15. In some situations, it may be desirable to have a larger window through which to view a particular history. For example, in Lucid, it is possible to write

$$\textbf{next}\ x \;=\; x + \textbf{next}\ \textbf{next}\ y$$

which requires a window on y as shown in Figure 4.16. The subset of Lucid compiled by Hoffmann's implementation [Hof78] permits such constructs. In this subset of the language, as in Lucid itself, **first** and **next** are treated as operators, and can be used in much the same way as the usual arithmetic and relational operators. The effects of this on implementation strategy are now considered. It can be seen from Figure

4.16 that, to generate code for a loop which includes both x and y, three values of y must be retained for use in the loop body. This can be determined from additional analysis of dependencies between variables. Further, it may be necessary to evaluate some inductive variables for a few steps, before the main iteration commences. For example, in

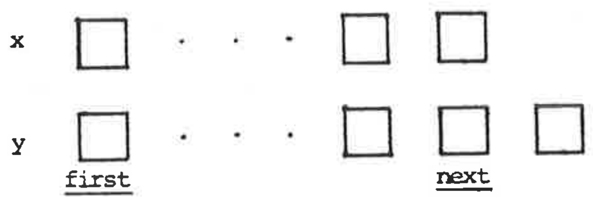


Figure 4.16. An extended window.

$$\textbf{first } x \;=\; \textbf{first next } y$$

y_1 must be evaluated before x_0 can be computed. Hoffmann's analysis involves the introduction of mappings which map expressions into "extended terms", in which occurrences of **first** and **next** have been replaced by qualified names. These qualified names are used as variables in the generated code, and thus provide the extra storage needed to implement the expanded window of, for example, Figure 4.16.

CHAPTER 5

A DEMAND DRIVEN IMPLEMENTATION OF LX

5.1 Introduction

In the implementation described in Chapter 4, an LX3 variable is identified with the sequence of values which flows along a particular arc of a data flow graph. In LX3 variables, are defined in such a way that each is seen as part of a loop, and successive values of each can be computed on successive iterations of the loop.

There are two principal deficiencies in this loop-based approach. Firstly, Lucid and LX programs may use non-pointwise transformations of histories; it has been pointed out, in §2.3.2.2, that definition of such transformations in LX3 is impracticable. The second is the problem of redundant computation (§1.8); the data flow graph generated from an LX3 program does not compute the minimal solution [AshW76] of the program. One reason for this is that history values are computed pointwise, that is one history element per loop iteration, even if the value is not used at that iteration. Another follows from a basic characteristic of data flow, namely that computation is driven by the availability of data, an evaluation strategy which permits highly parallel program execution, but at the expense of sometimes computing values that are subsequently discarded.

A data flow implementation based on the demand driven operational semantics given in §2.3.1 overcomes both of these difficulties. In this chapter, such an implementation of LX is described, for the data flow interpreter described in Chapter 3. The chapter concludes with a discussion of possible improvements to the implementation scheme.

The steps in the translation of a complete example LX program are presented is some detail in Appendix 4.

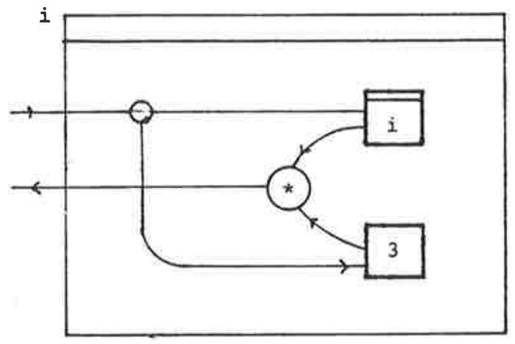
5.2 A simple example

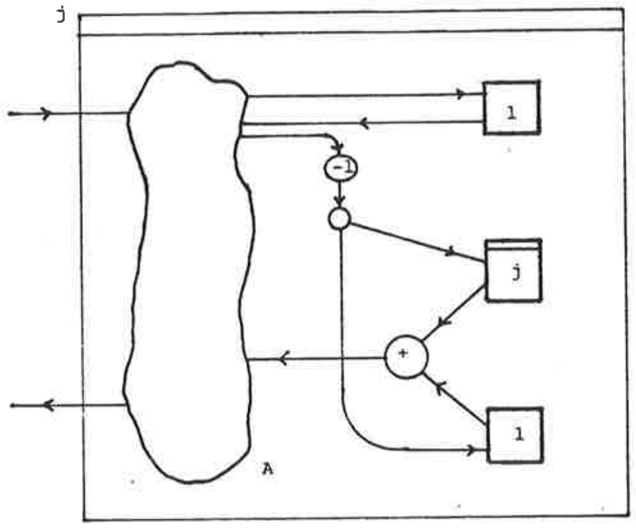
In compiling an LX program into demand driven data flow code, the following method is used. Firstly, each definition of the program is compiled into a data flow network. A function template is created from each such network; each template is defined with one parameter, a demand, and one result, a value. A use of a definition is compiled as the application of the function template corresponding to that definition; this creates an activation of the template with an appropriate demand as parameter. The purpose of a function activation is to compute a single value (rather than a history) corresponding to the demand it receives. To do this, it will often demand values from other definitions; it has been shown, in §2.3.1.1, that the demand number of every such value can be derived from the incoming demand number. New activations are thus created as needed, with one activation for every value needed.

These ideas are illustrated in the translation of the following three definitions.

```
i =  1 fby i+1
j =  3 * i
r = asa i eq 3 then j easa.
```

The function templates shown in Figure 5.1 are generated from these definitions; in Figure 5.1 and elsewhere, function templates are represented as in Figure 5.2, with one arc entering, carrying a demand, and one leaving, for transmission of the demanded value.





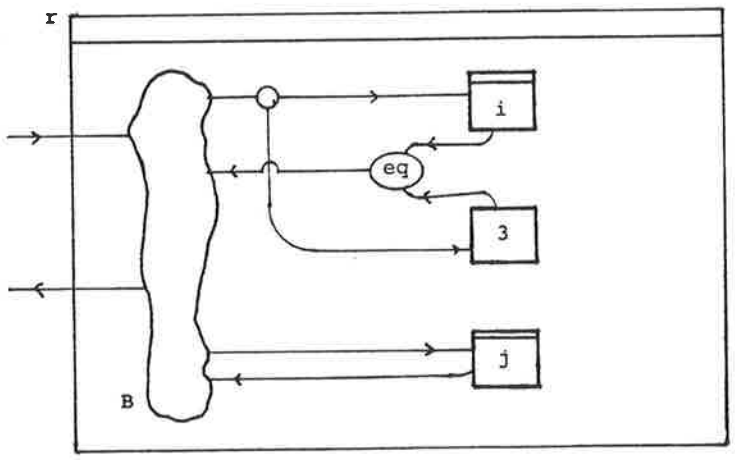


Figure 5.1. Function Templates for variables i, j and r.

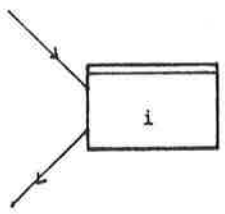


Figure 5.2. Representation of a function template.

Consider the template for i. The area labelled A represents a data flow network which transforms the incoming demand d according to the DT for the operator **fby** given in §2.3.1.1.3; it also returns the value received in response to the demand. In the template for j, the demand number is used, unchanged, to activate a computation of i; this is consistent with the DT for *, a pointwise operator. The area labelled B in the template for r, implementing the DT for **asa_then_easa** (§2.3.1.1.3), issues demand numbers 0, 1, 2, and so on, to the network representing the condition of the **asa** definition, until the value *true* is returned; the corresponding demand number is then sent to the expression component of the **asa** definition. The value yielded in response to this demand is returned directly as the result of the template for r.

The execution of this program fragment is now traced. Execution commences with the arrival of a demand (from a source external to the program) for $[\![r]\!]_0$, which triggers the creation of a function activation from the template for r, and transmission of the demand to that activation. The variable r is defined using an **asa** definition, which requires that values of its condition be demanded until a value *true* is returned. Hence, a demand number of 0 is then propagated to both the function application i and the constant instruction "3". The latter is enabled immediately, and fires to produce a value 3 at an input of the EQ instruction. The function application causes an activation to be created from the function template for i, to which the demand number 0 is transmitted. As shown above, the demand will in turn be propagated to the constant "1" (the left operand of the **fby** operator in the definition of i), the value 1 returned from the activation, and propagated to the EQ instruction. The comparison fails, so a demand number of 1 is transmitted to the FT for i, resulting in the creation of another activation of i, termed i_1. Within i_1, the right hand operand of the **fby** is chosen, and a demand number $1 - 1 = 0$ is propagated to both the

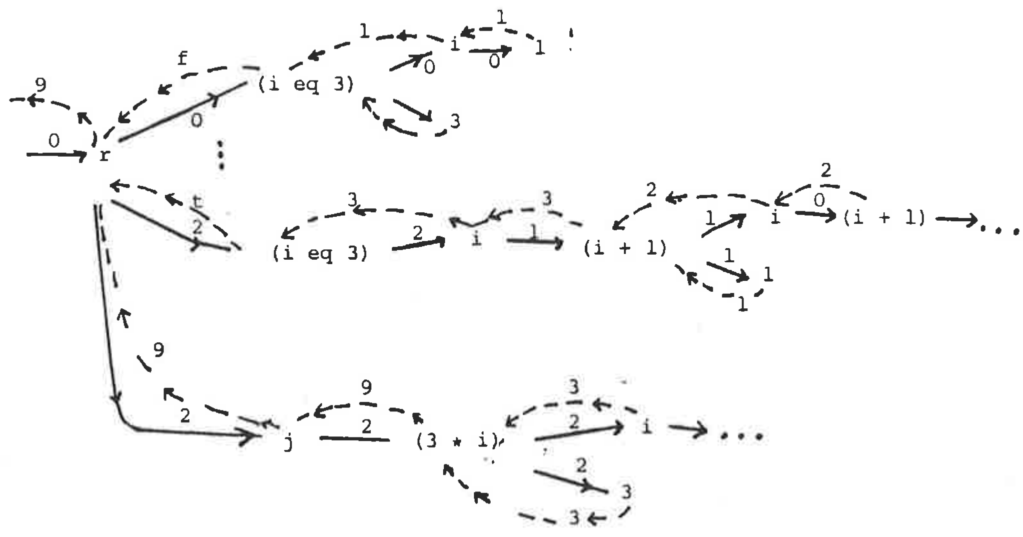


Figure 5.3. Demand flow in the example of Figure 5.1.

constant instruction and a further activation of i, i_0. The activation i_0 returns the value 1, as described above, the addition instruction fires, and the value 2 is returned from i_1. The comparison again fails, another activation of i is created with a demand number of 2, producing the value 3. The test now succeeds, a demand number 2 is transmitted to an activation of j, and propagated to yet another activation of i. A value 9 is eventually produced from the activation of j, and returned as the result of the program. Figure 5.3 illustrates the flow of demands and the return of values.

It is apparent from this description that, although the implementation does not attempt to compute any elements of histories which are not required to determine the solution of a program, recomputation (§1.8) of particular elements occurs. In §5.4, a means of reducing the extent of this recomputation is discussed.

5.3 The implementation

The implementation consists of a compiler and a data flow interpreter. The compiler analyzes the program source, producing an initial heap which includes function templates and appropriate tables. The initial heap provides the initial configuration for the execution of the program by the data flow interpreter.

In the next section, the representation in data flow of the principal components of the semantic model of an LX program (§2.3.1) is described. The translation schemes used for LX language constructs are presented in subsequent sections.

5.3.1 Representation of Demands and Environment Tables

The principal structures of the semantic model, defined in §2.3.1.1, are the demand and the environment table (ET). The data flow interpreter directly supports binary trees as objects on the heap, with instructions for the manipulation of trees. Demands and environment tables are therefore represented as binary trees.

A demand is represented by a tree, the left branch of which is a demand number, and the right, the tree representation of an instance, defined (§2.3.1.1) as the pair $\langle$ *Text, ET* $\rangle$. The textual component of an instance is represented by the Uid of a function template (FT); any such textual reference can be satisfied by using the Uid of a template on the initial heap. An instance is also represented by a tree, the left branch of which is the Uid of the appropriate FT, while the right branch is the tree representation of an ET.

An ET has two components, namely *FreezeVal*, an index at which histories are to be frozen and *List*, a list of identifer associations. Again, an ET is represented by a tree, the left branch of which is *FreezeVal*, and the right the tree representation of *List*. Each list element, defined in the model as

$\langle$ *Ident, User, NewIdent* $\rangle$,

is also represented by a tree, with the following structure:

```
             ...
            /   \
           /     \
          /      ...
      Uid1      /   \
               /     \
           Inst1     Uid2.
```

Uid1 and *Uid2* represent *ident* and *NewIdent*, respectively; an identifier is represented by the Uid of the function template generated from the text which defines the identifier.

The *User* component is represented as an instance *Inst*1, the representation of which was described above.

The data flow interpreter supports primitive operations for the manipulation of trees. These operations can be used to define higher level operations appropriate to the structures of the semantic model. The following operations have been defined in the interpreter, but can be expressed in terms of the more primitive operations on trees:

DCons	construct a demand from a number and an instance
DNum	select the Num component of a demand
DInst	select the Inst component of a demand
DNumD	decrement by 1 the Num component of a demand
DNumI	increment by 1 the Num component of a demand
DNum0	set the Num component of a demand to 0
ICons	construct an instance from a Uid and an ET
IFT	select the Uid component of an instance
IET	select the ET component of an instance
ETCons	construct an ET from a FreezeVal and a List
ETFrz	select the FreezeVal component of an ET
ETList	select the List component of an ET
LCons	construct a list element from 3 input components
LAppend	append one list to another
LHd	select the element at the head of a list
LTl	select the tail of a list
LIdent	select the Ident component of a list element
LUser	select the User component of a list element
LNewId	select the NewIdent component of a list element.

In these definitions, "list" is a list of identifier associations of the form described above. Much of the information about the formals and globals of a clause needed for the *List* component of an ET can be determined by the compiler; for example, the *Ident* component of a *List* is known at compile time. It is thus convenient to assume that the initial heap contains structures representing partially completed lists, and to define the following operations, which permit a concise expression of the list manipulations required by the model:

LUpd	takes a list *L* and an instance *i* as arguments, and produces a list in which the *User* component of each element of *L* is replaced by *i*
LFind	takes a list *L* and the Uid of an FT *id*, searches *L*, and returns the entry whose *ident* component is *id*.

5.3.2 Representation of primitives

Many of the primitives of the semantic model are concerned with information which can be derived at compile time. The primitives *Class*, *Frozen* and *Subject* are in this category, as are the lists of identifiers *Formals* and *Globals*, and the list *Actuals*. The primitives *CreateInstance*, *CreateTable* and *Table* are implemented using the operations on structures defined above; as indicated above, some structures can be partially constructed at compile time. Details are given below of how the compiler uses available information in constructing objects on the initial heap.

The representation of the **transmit** primitive depends on the context in which it is used. For example, the **transmit** primitive in the DT for a program §2.3.1.1.1) uses the **envof** option and transmits a demand to the definition of *result*. The implementation of the primitive requires the construction of a demand, using the *DCons* operation, and the application of the function template for *result*. In the case of an expression, the representation of **transmit** depends on whether the demand is transmitted to a node or to a leaf of the expression tree. Details of the representation in a given context are included as part of the descriptions of function template construction which follow.

5.3.3 Compilation of program constructs

5.3.3.1 Program

As in Chapter 4, a single pass, recursive descent compiler is used, but no dependency analysis is performed by this implementation. Compiler actions can be related directly to the productions of the definition of LX syntax (Table 2.1); in the case of PROGRAM, the compiler firstly records type and other information about the program name, the globals and the frozen variables. It also constructs, on the initial heap, a partially completed list of global identifiers, in the manner described in §5.3.1. For the globals of the program, special function templates are constructed, which incorporate facilities for communication with the external environment. Each definition associated with the program is then compiled.

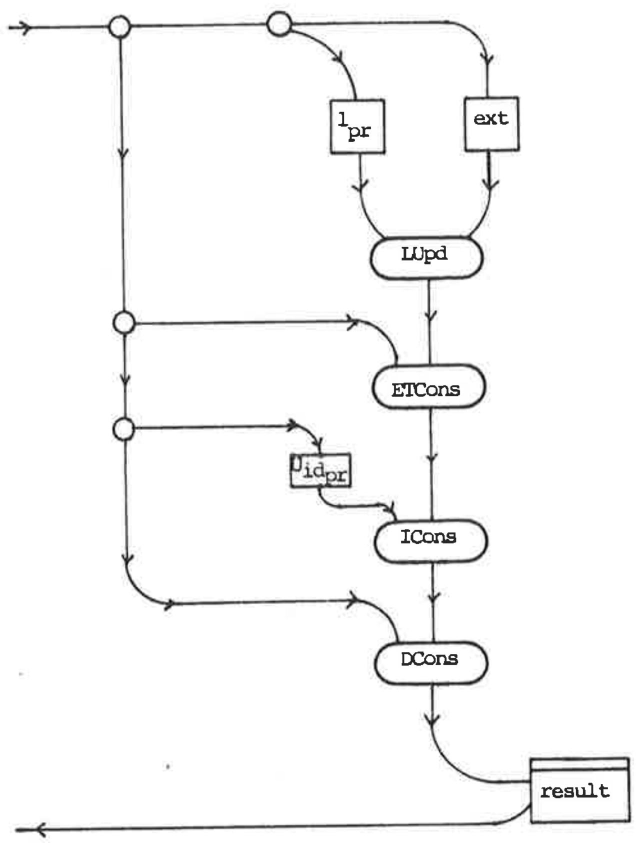


Figure 5.4. Function template scheme for PROGRAM.

Finally, the function template for the program itself is constructed, using the data flow scheme shown in Figure 5.4, in which the value arriving from the left is the demand which triggers the program. In the figure, l_{pr} is a constant, the Uid of the aforementioned list of global identifiers constructed by the compiler. The constant *ext* is the Uid of an instance used to represent the external environment. The constant Uid_{pr} is the Uid allocated by the compiler for the function template of the program itself; it represents the text of the program within the instance which is created by *ICons* and used in the propagated demand. The latter is shown as being transmitted to the definition *result*; this is an abbreviation for the application of the function template for *result*, the expansion of which is given in §5.3.3.4.

5.3.3.2 Definition

From the syntax definition of LX (Table 2.1), it can be seen that there are three categories of DEFN, namely a declaration, an equation, and a **define** clause. The actions taken by the compiler in analyzing an LX definition are now described; they are determined by the requirement to maintain internal information about the program, and the necessity to implement the specifications of the DT.

A declaration specifies the type of an identifier, and hence the compiler need perform no code generation actions, but simply records appropriate information in its symbol table.

Consider the case of an equation. Given that the primitives *Class* and *Subject* can be evaluated by the compiler, it can be seen that the DT for a definition entry §2.3.1.1.2) specifies that an incoming demand be propagated to the right hand side of the equation. The compiler is thus required to construct a function template from the expression on the right hand side of the equation; the method used to do this is considered in detail in the next section.

Finally, consider the compilation of a **define** clause, the general form of which is as follows:

```
define SUBJECT [ PARAM { , PARAM } ]
      [ using IDENT{ , IDENT } ]
      [ freezing FREEZE_LIST ]
      DEFN_LIST
edefine.
```

The subject identifier is entered in the symbol table to enable evaluation of the predicate *Subject* when required. A list of formal parameters is added to the initial heap, for use, as described above, in constructing an ET. As described in §2.3.1.1.2, a definition entry exists for each identifier accessible within a clause; accordingly, a function template must be provided for each such identifier. The case alternative "formal, global" of the DT for a definition entry specifies how the template is to be constructed. For example, if the formal parameter x is not frozen, the function template for its definition entry in the clause will be constructed according to Figure 5.5. The constant Uid_x in Figure 5.5 is the Uid which represents the identifier x. In this template, the transmission of the demand is implemented by invoking the function template identified by the table entry, with the newly constructed demand as its parameter. If x is frozen,

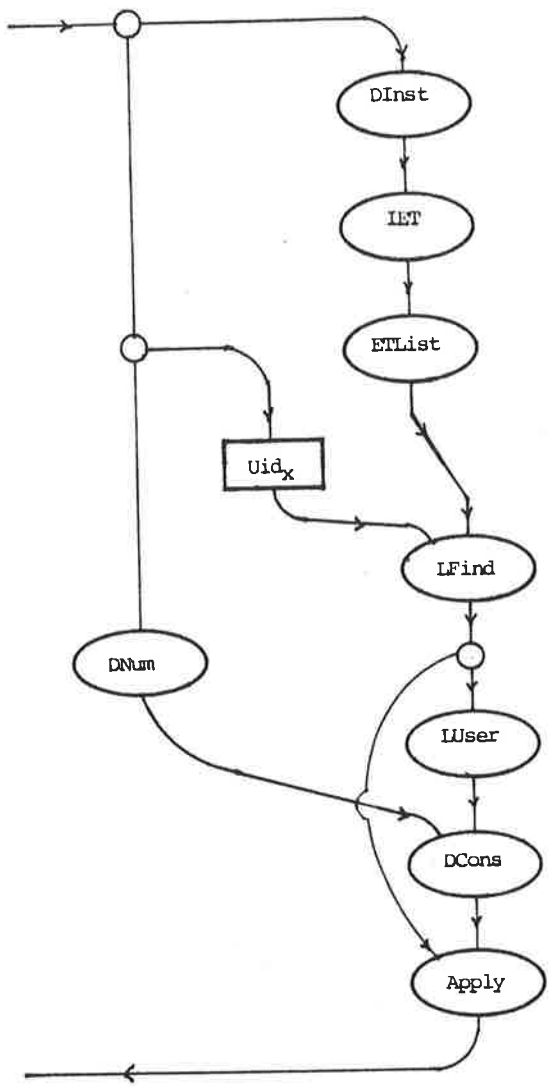


Figure 5.5. Function template scheme for an unfrozen formal parameter.

the instruction *DNum* is replaced by a network which selects the *FreezeVal* component of the ET associated with the incoming demand.

The **using** list specifies the globals associated with the **define** clause. The action taken by the compiler for each identifier on this list is the same as for a formal parameter of the clause. In the case of the **freezing** list, sufficient information about each entry is recorded in the symbol table to permit the compiler to evaluate the primitive *Frozen* when required.

After compiling each entry in the definition list, the compiler constructs a function template for the subject of the clause, as specified by the DT (§2.3.1.1.2). The scheme used is almost identical to that presented for the main program, except that the list input to the *ETCons* operator, as shown in Figure 5.6, is formed differently. In this figure, the lower input is the demand which triggers the function template; it is used also to trigger the constants l_{fp} and l_{gl}, Uids of the partially constructed lists of formals and globals, respectively. The input *ui* is the Uid of the instance in which the use of the clause occurred, and is transmitted from the point of use. A list of Uids of function templates which compute the values of actual parameters (the list *Actuals* of §2.3.1.1.2) is also constructed at the point of use, and included with the demand (also see §5.3.3.4).

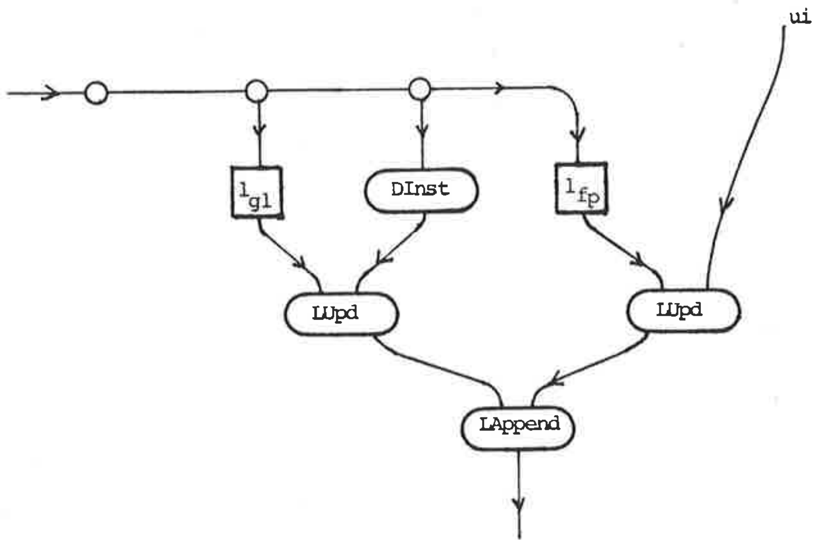


Figure 5.6. Data flow scheme for the list component of an ET.

5.3.3.3 Right-hand-side and expression

Here, as in §2.3.1.1.3, the syntactic notions RHS and EXPR will be considered together as expressions, as both consist of operators and operands. The compiler constructs a function template from each such expression, which again is assumed to be in tree form.

The root of an expression tree can be viewed as the operator of an expression, and the branches of the tree as operands. The DTs given in §2.3.1.1.3 specify the transformations appropriate to each operator; each DT can be represented as a data flow network which transforms an incoming demand. A data flow network which accepts the transformed demand and returns the demanded value is associated with each operand. As each operand is in turn made up of operators and operands, this construction is applied recursively to determine the form of the function template. The recursion terminates when an operand is either a constant or an identifier; the action taken by the compiler in this case is described in the next section.

It should now be clear that the general pattern of Figure 5.7 can be used in constructing a data flow network for any ⟨*operator, operand(s)*⟩ pair. The area labelled B in the figure is a data flow network constructed using the appropriate DT, with any additional data flow instructions required to ensure that the values demanded are returned correctly; the return of values is considered in more detail in §5.3.3.5. The boxes o_1 and o_3 represent the operands of the expression.

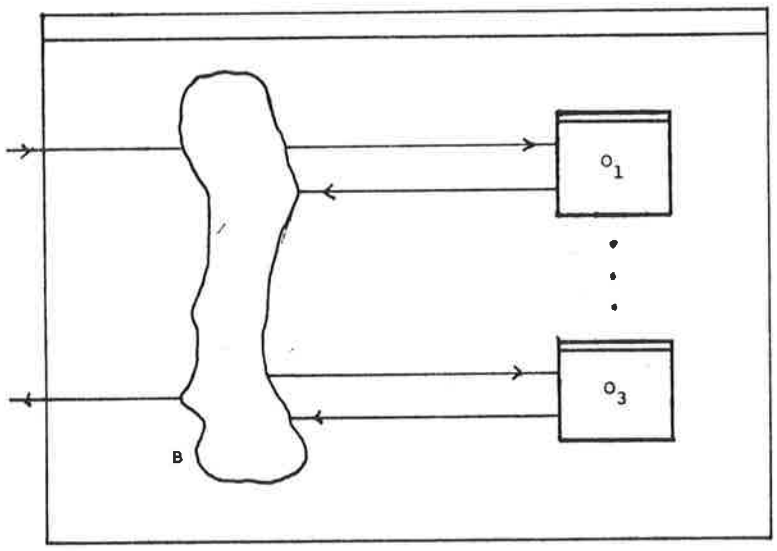
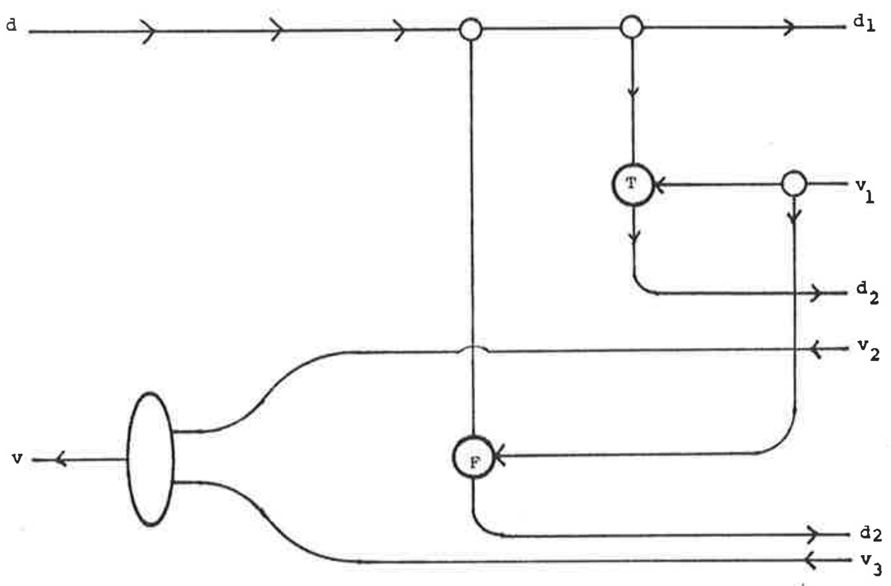


Figure 5.7. A general data flow scheme for expressions.

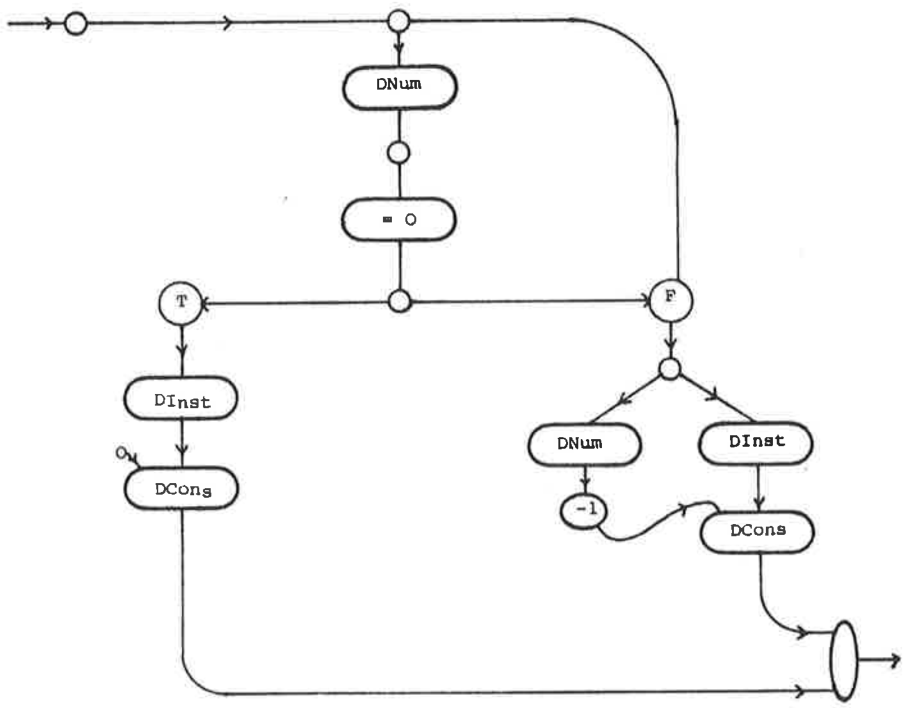
To determine the data flow network which implements the demand transformation required in the area labelled B in Figure 5.7, it is necessary to associate a data flow network with each of the operator DTs of §2.3.1.1.3. As examples, the networks associated with the operators **if_then_else_eif** and **fby** are given in Figure 5.8.

5.3.3.4 Identifiers and literals

The previous section described the analysis of an expression tree, and associated



(a) conditional (all control arcs are from v_1)



(b) **fby** (return of values not shown)

Figure 5.8. Data flow schemes for conditional and **fby**.

compiler actions. Identifiers and literals are encountered as leaves of the expression tree. In the case of a literal, the demand can be satisfied, and it is not necessary to propagate it further; compiler actions associated with the return of the value are described in the next section.

In the case of an identifier, the compiler generates code which causes invocation of the function template associated with the definition entry for the identifier. Firstly, consider an identifier *i* used with no parameters. In §5.2, a use of such an identifier was represented as the invocation of the corresponding function template using the scheme shown in Figure 5.2. This scheme is now regarded as an abbreviation for the more detailed scheme shown in Figure 5.9, in which Uid_i identifies the function template associated with *i*, which, as noted in §5.3.3.2, can always be determined by the compiler and inserted as a constant in the generated code.

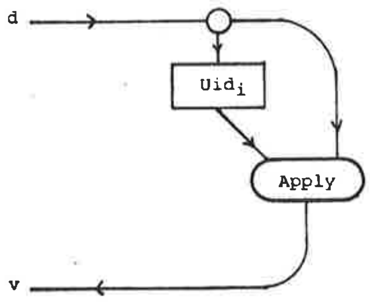


Figure 5.9. Data flow scheme for a use of an identifier.

Essentially the same scheme is applicable when *i* is used with actual parameters. As discussed in §5.3.3.2, information about actual parameters and the instance in which the use of *i* occurs, must be transmitted to the function template Uid_i, where such information is necessary for constructing an ET; however, the scheme must be modified slightly for this purpose. Perhaps the simplest way is to suppose that the information is carried with the demand; the scheme can then be used unchanged, but the representation of a demand redefined. An alternative is to pass the information as an additional parameter to the APPLY instruction. The former approach has been implemented, but further details are not given.

5.3.3.5 Return of values

The model of §2.3, and the descriptions of previous sections, are concerned principally with the propagation of demands and the construction of data flow networks which implement demand transformations. The value return path, or VRP, is data flow code which enables the return of demanded values; this section is concerned with the construction of such paths by the compiler.

When a demand is propagated to a literal, the demand is satisfied and the value of the literal can be returned to the point of origin of the demand. The compiler implements this by representing the literal as a CONSTANT instruction, and the first stage of the VRP as an arc directed to a point in the function template which can either use the value, or continue its transmission along the VRP. The arc can always be constructed using statically determined information; either the demand came from within the function template, or the literal is the only component of the expression, and hence of the function template. In the latter case, the value can be returned as the result of the function template, whereas in the former, the literal must be an operand of an operator, and the value can be directed to the network associated with that operator.

In the construction of a function template from an expression, it is clear that values can be regarded as returned from the leaves of the expression tree. The operator of which the leaf is an operand contributes to the VRP with some transformation of the value returned to it from the leaf node. For each operator, a definition is now given of the transformation required to describe how the value or values returned from each operand are to be used in determining the value v propagated on the VRP in response to the original demand d. In the definitions which follow, v_i is defined as the value returned from operand i in response to demand d_i (see Figure 5.7):

arithmetic and relational operators	$v = v_1$ op v_2
unary operators	$v =$ op v_1
first	$v = v_1$
next	$v = v_1$
fby	**if** *d.Num*= 0 **then** v=v_1 **else** v=v_2
if_then_else_fi	**if** v_1 **then** v=v_2 **else** v=v_3
asa_then_easa	v=v_2
wvr_then_ewvr	v=v_2
upon_then_eupon	v=v_2.

Data flow networks constructed from these definitions are included in function templates at appropriate points.

A VRP segment must also be constructed at those points in a function template which correspond to the use of the **transmit** primitive in the DTs for PROGRAM and definition entry; for example, within the latter, the statement

transmit *d* **envof** *x1* **to** *result*

clearly implies that a value will eventually be returned from the function template corresponding to *result*. In this and similar cases, the value returned can be considered the result of this function template, and an appropriate arc is constructed by the compiler.

5.4 Discussion

Three significant achievements emerge from the work of this chapter. Firstly, the semantic model itself has been validated by examining the execution of a representative sample of LX programs. Secondly, it has been shown that the operational semantic model of LX (§2.3.1) can be used in conjunction with a recursive–descent based translator to produce quickly a correct implementation of LX, thus demonstrating the usefulness and importance of a precise operational model. Thirdly, it has been shown that demand driven computation can be expressed using purely data driven schemes, and used to overcome the problem of redundant computation.

A characteristic of the implementation is the "interpretive" nature of the code generated, in the sense that the target language is effectively a high level virtual machine which incorporates directly many of the primitives of the semantic model. While this is advantageous for developing a prototype implementation, such as described in this chapter, it is inefficient. There are two possible directions for improvement in this area. Firstly, the virtual machine could be refined so as to be closer to specific data flow machines. A second interesting possibility would be to design a physical realization of the virtual machine itself.

The implementation potentially increases parallelism in that it spawns many activations in response to the propagation of demands, with each such activation exploiting the parallelism inherent in data flow. A significant deficiency of the implementation is that many of the activations so spawned attempt recomputation of the same history element. A similar effect was noted by Maurer and Oldehoeft [MauO83] who, in translating a purely functional language to data flow graphs, treat a structure element as a function, and observe that although such a function is not evaluated until it is applied, such evaluation must occur each time the element is accessed.

It is clearly essential that steps be taken to alleviate this problem of recomputation. An effective method of eliminating the necessity to recompute a value is to store that value the first time it is computed. The possibility of introducing such a notion of storage into the semantic model and the implementation is now discussed.

It is proposed that storage be provided for each variable in the main program, and dynamically for each use of a clause subject; it is understood that use of a subject gives rise to an instance of its defining clause in which storage is similarly associated with variables and uses of subjects. In the existing semantic model (§2.3.1), an instance of a clause is created for the computation of a single value. In extending the model in this way, an instance becomes a repository for the values contained in all associated histories, which are included in the entries of the instance's ET.

Facilities must be provided to detect the first demand on a history, to create associated storage, to add and retrieve values, and to release storage. The first two are illustrated by considering the main program. Assuming that a heap entry for an instance of the main program is pre-allocated, the DT (Figure 2.4), and the FT which implements it (Figure 5.4), can determine whether or not the instance has been entered previously, and, if not, initialize it; propagation of a demand for the first time to a definition entry within the instance causes allocation of storage for that entry. Instances which create other instances can use similar techniques to detect the first demand propagated to an instance.

Values can be added to storage at their site of computation; in implementation terms, an "append" operation can be inserted at a suitable point on the VRP (§5.3.3.5). Retrieval of values from storage can be arranged by intercepting a demand before it is propagated to a particular definition entry, and checking the storage associated with that entry to decide if it is necessary to transmit the demand.

The release of storage is a little more complicated. Each instance is now responsible for a history, not a single value; in the latter case, the instance and its storage can be deallocated when the value is returned (exactly as for a function in a conventional language). As a history is potentially infinite, it is usually necessary to perform some program analysis to find situations in which storage can be released. For example, a quiescent definition (see §2.3.2.2) can be replaced by a single value. Program analysis could be used in many cases to provide bounds on the amount of storage required.

An important aspect of compilation is optimization, which was not considered at all in §5.3. Some possible optimizations are now discussed.

If a notion of storage is introduced as suggested above, various storage optimizations are possible. As already mentioned, it may be possible to determine bounds on the storage required for some histories. If the same definition is used in several clauses, it should be possible to share storage by using one copy of the definition. Some use of memo functions [Mic68] is possible; this is the technique of associating a table of argument values and function results with a function definition, to reduce recomputation. Clearly, LX clauses can generally not use this technique, because the arguments are infinite histories. However, a clause with all its parameters and globals frozen acts exactly as a conventional function, and the technique is then appropriate.

The implementation performs considerable manipulation of demands; optimizations which reduce demand handling in appropriate situations should be investigated. Similarly, it could be arranged in many cases for values, particularly constants, to be returned directly to the original point of demand, rather than following the complete VRP.

Machine specific optimizations should also be explored. For example, techniques appropriate to the Manchester machine for balancing the data flow graphs of expressions, and for common subexpression elimination have been investigated by Jones, Kidman and Morello [JonKM85].

CHAPTER 6

A HYBRID IMPLEMENTATION SCHEME

6.1 Combining the data flow and demand driven schemes

The implementations of two different strategies for the translation of LX programs to data flow graphs have now been described in Chapter 4 and in §5.3 respectively. As mentioned previously, the former is restrictive in that it can compile only a certain class of programs corresponding to the subset LX3, but involves a straightforward transformation into code "in the data flow idiom"; in fact in many ways, an LX3 program can be viewed as a high level representation of a data flow machine-level program. The latter is general in that it can translate any LX program to data flow graphs, but the graphs are somewhat "forced" in terms of their adherence to a natural data flow idiom. Stated another way, an LX program can be represented as a data flow graph which is most suitably interpreted on a "virtual machine" somewhat removed from a realistic data flow machine. Consequently, it is inefficient as a program for a data flow machine.

Such a data flow program can, of course, be made more efficient by applying any of several optimization techniques, for example, those discussed in §5.4. However, a different approach is explored in this chapter. It is proposed that the two strategies mentioned above should be combined in such a way that the networks of Chapter 4 are used except when demand driven computation is required by the source program (for example, to give control of input and output, as discussed in Chapter 7). This hybrid approach complements the use of optimization as a means of producing more efficient target language programs; it can also be seen as increasing the expressiveness of the language LX3.

Extensions to the language LX3 are presented which, subject to appropriate restrictions, enable programs optionally to specify a demand driven component as part of the main program or any **define** clause; the definitions of this component are to be implemented in the demand driven manner of §5.3, whereas other definitions are to be translated using the techniques described in Chapter 4. The implementation of the interface between the two components is discussed, and an example of program translation is presented. Finally, possibilities for widening the domain of applicability of the scheme are discussed.

6.2 Language extensions

Firstly, some terminology is introduced. Two sets of identifiers, termed DF and DD, are associated with the main program, and each **define** clause. The definition of each identifier in DD is to be translated using demand driven techniques. All other identifiers are in DF, and their definitions are to be translated into data flow networks, referred to as the *data flow component*, DFC, of the clause. Similarly, the networks which result from the translation of the definitions of identifiers in DD comprise the *demand driven component*, or DDC, of the clause. Occasionally, the notation DF_z is used to denote the set DF associated with a particular clause subject z; DD_z, DFC_z and DDC_z are used similarly.

The following rules describe the proposed language extensions, and the circumstances in which they are applicable.

(1) The main program identifier is in DF.

(2) Variable declarations within a clause whose subject z is in DF (including the main program clause) may be specified as **in dd**; in this context, the BNF definitions of DECLARATION and GLOBAL given in Table 2.1 are replaced by the following:

DECLARATION	::=	TYPE IDENT [**in dd**] { , IDENT [**in dd**] }
GLOBAL	::=	TYPE VAR **in dd**

Within such a clause, an identifier specified as **in dd** is included in DD_z, otherwise in DF_z. In general, the formal parameters and global variables of z are included in DF_z, with the exception of variables specified as global to the main program, which are included in DF_{main}.

All identifiers declared within a clause whose subject z is in DD are included in DD_z; DF_z is empty for such clauses.

(3) The definition of an identifier x in DD_z may be written according to LX syntax, using identifiers from both DF_z and DD_z, provided that the definition of x does not use any identifier y in DF_z which is the subject of a parameterized **define** clause.

(4) The definition of any identifier x in DF must conform to the syntax and restrictions of LX3, except as modified by these rules. DD identifiers may be used in the definition of x; in the case when x is a subject, any DD subject inherited by x must freeze all its parameters and globals.

(5) Consider an identifier y in DF_z, used by the definitions of one or more identifiers in DD_z. These definitions must be written in such a way that DDC_z does not issue any demands for values of $[\![y]\!]$ which are in the "future" of y relative to the current state of DFC_z.

Rule (1) ensures that the main program is regarded basically as an LX3 program extended to have both a DFC and a DDC. Note that the declarations of the locals and globals of the main program determine DF_{main} and DD_{main}; DF_{main} does not include the main program identifier itself, which is best regarded as a DF identifier external to the program.

Rule (2) ensures that DF_w is empty for a subject w in DD, so that only LX3-based clauses have both a DFC and a DDC. An LX3-based clause z has frozen global variables and parameters; in effect, the environment which satisfies the definitions of the clause is determined in terms of these frozen (constant) values, and the definitions of the local variables of the clause. An important consequence of this is that demand flow within DDC_z will always be confined to identifiers which are either local to z or have a constant value within z. In other words, it is never necessary to propagate a demand outside z, hence DDC_z requires no information about the environment global to z, a fact which greatly simplifies the task of communicating environment information between the components.

Although the main program is LX3-based, Rule (2) stipulates that its globals are included in DD_{main}, which ensures that input from the external environment has a demand driven semantics. There are two advantages in using such semantics. Firstly, only those input values actually required by the program need be supplied to it; data driven input semantics may require input values to drive a part of the computation which simply discards the values. Secondly, because an input value can be supplied in response to a demand, or request, from the program, a dialogue can be established with the program. See §7.4.4 for further discussion of this topic.

Rule (3) ensures that identifiers in DF_z can be represented uniquely in DDC_z, and that the required representation can be determined statically. In §6.3, it will be seen that a variable y in DF_z can be represented in DDC_z by a function template which accesses a structure containing values from $[\![y]\!]$. If y were a parameterized subject, one such structure would be required for each use of y in a definition of an identifier in DD_z; the number of such uses cannot be determined statically in the presence of recursion. A consequence of Rule (3) is that data flow procedures in DFC_z are invoked only from DFC_z.

Consideration of Rule (4), in conjunction with Restriction (4) of the definition of LX3 (§2.3.2.1), leads to the conclusion that *result* must be in DF, and part of a loop. Hence, each LX3-based clause has a loop structure determined by an **asa** definition; this ensures that the techniques of dependency analysis and loop synthesis described in §4.3 remain applicable to the extended language. A DD subject which freezes all its globals and parameters can be inherited, as such a subject is defined in terms of values known to the DFC; similarly to an LX3 clause, it can be regarded as a subcomputation of an outer loop (§2.3.2.2).

Rule (5) is a restriction which cannot be checked statically, but is imposed to prevent deadlock. The rule is analogous to Restriction (5) of §2.3.2.1, and prevents DDC_z from attempting to access any values which are yet to be computed by the DFC. Further explanation is given in §6.4.

6.3 Schemes for the extended language

In this section, schemes used in the construction of the DFC and the DDC are described. For the cases in which either a clause subject is in DF, and DD for the subject is empty, or the subject is in DD (hence, all its local identifiers are in DD), the schemes of Chapters 4 and 5 respectively, apply unchanged. Consequently, schemes are considered only for a clause whose subject is in DF and which contains both a DFC and a DDC.

As described in Chapter 4, the data flow program produced from an LX3 program has the following characteristics. Firstly, there is a function template (FT) for the main program, and for each **define** clause. Each such FT is composed of interconnected circulators, synchronized by control values generated by the termination condition network. The circulators generate the values in the history of an LX3 variable consecutively, on the assumption that each value can be computed from previous values.

In contrast, an LX program is translated into an FT for each definition. An FT is invoked, with a demand as parameter, for each required value of a history.

The construction of the networks of the DDC is essentially the same as described in §5.3, except that a different construction is needed for uses of identifiers in DF; details will be given later. Attention is now given to the design of the interface between the DFC and the DDC, and the schemes used in the DFC to implement this interface.

The interface between the DFC and the DDC serves two important purposes. Firstly, by storing values computed by the DFC, it enables the DFC to service demands issued from the DDC. Conversely, it permits the DFC to transmit demands to, and receive values from, the DDC.

These objectives can be achieved as follows. A history structure is built within DFC$_z$ for each variable in DF$_z$ which is required to supply values to DDC$_z$. A special FT is generated within DDC$_z$ for each such DF variable; this FT interprets a demand for a value as an access operation on the appropriate history structure. Further details are given in §6.3.2.

The history structure is represented in the data flow model as an early completion data structure (ECDS) [Den81]. An ECDS can be used before it is fully defined; this property is used here to separate the actions of creating, writing to and reading from the structure. The undefined structure is created initially in the DFC, and transmitted to both the DFC, which generates its component values, and to the DDC, which accesses the values as dictated by the flow of demands. See §6.3.1.2 for more details.

The history structure is the mechanism used by the DFC to supply values to the DDC; a request from the DDC for a value computed by the DFC causes the structure to be accessed. The DFC obtains a value from the DDC by transmitting the current DFC iteration number to the DDC as a demand number, as described in §6.3.1.1 below.

6.3.1 DFC schemes

The schemes used to construct the DFC are essentially those presented in §4.2; additional schemes required for the interface to the DDC are now described as modifications of those presented in Chapter 4.

6.3.1.1 Construction of a demand and invocation of a DDC FT

Consider a clause with subject z in DF, in which the definition of some variable in DF_z uses an identifier x in DD_z. DFC_z constructs a suitable demand, and uses it to invoke the relevant FT in DDC_z. The required demand number is the iteration number of the loop from which the demand is transmitted. The iteration number is made available by including in DFC_z a circulator corresponding to the definition

$$i.n \;=\; 0\ \mathbf{fby}\ i.n + 1.$$

A demand also contains an instance component, which is used to pass environment information to an FT. However, it is unnecessary for the demand entering the DDC from the DFC to carry any such information because, in fact, there is no global environment to be carried forward. This can be understood as follows. The ET of an instance is used to resolve uses of global identifiers and formal parameters in terms of instances through which the demand has passed in arriving at the current instance; in effect, each such previous instance contributes to the current environment. Because the clause subject z itself is in DF, the parameters and globals of the clause are frozen, and are represented in the DFC as simple circulators which make the single, frozen value available to each iteration, as in a local, constant definition. Consequently, the *instance* component of a demand entering the DDC from the DFC can be empty.

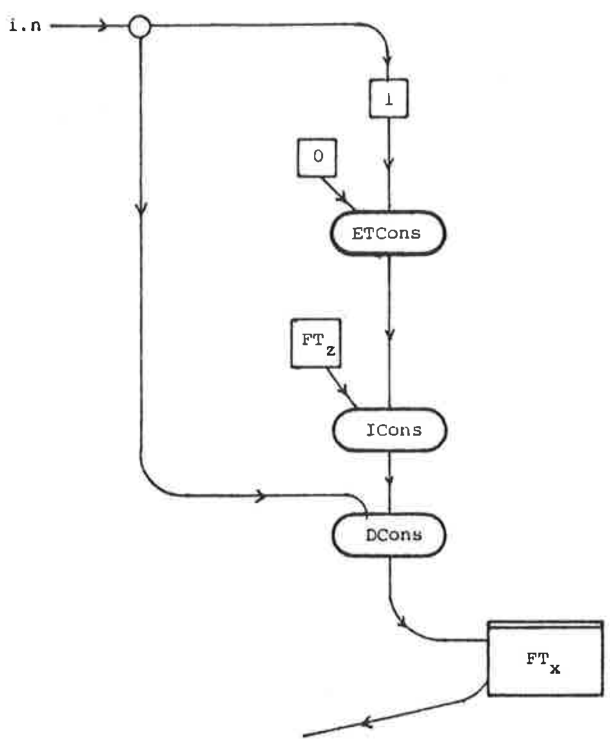


Figure 6.1. A scheme for the invocation of an FT in the DDC.

The FT representing x can be invoked from DFC_z according to the scheme shown in Figure 6.1, in which $i.n$ is supplied by the circulator described above, and FT_x is in DDC_z. A dummy instance is created; components of both the ET and the instance can be defined arbitrarily.

Consider the incorporation of this scheme into the LX3 translation scheme. It can be seen that the use of x is effectively a use of the DF inductive variable $i.n$. Consequently, when it is found necessary to create a link from x, links can be created instead from $i.n$; if the use of x is qualified with either **first** or **next**, values can be taken from the appropriate characteristic address of $i.n$. Similarly, dependency analysis and gate generation can be based on $i.n$.

6.3.1.2 Construction of history structures

Consider the case in which y is an identifier in DF_z, used within DDC_z. An additional circulator is included in DFC_z to generate a structure H containing values in the history $[\![y]\!]$. The history structure can be viewed as the tree shown in Figure

6.2, the root of which is the original structure allocated by the compiler, and in which Y denotes the history $[\![y]\!]$.

```
     .
    / \
  Y0   +
      / \
    Y1   +
        / \
      Y2   .
            .
             .
```

Figure 6.2. A history structure.

The circulator corresponds to the definition:

$$H \;=\; FirstH \;\textbf{fby}\; H +\!+\; y.$$

FirstH represents a network which adds y_0 to the initially undefined history structure for Y allocated by the compiler. The notation ++ indicates an operation which takes the current values of the history structure and Y and produces an updated history structure according to the scheme given in Figure 6.3, in which the operation PAIR creates a new, undefined ECDS. MKR inserts the newly created ECDS into the right component of H, while MKL sets the left component of the newly created structure to the current value of Y; the right component remains undefined.

6.3.2 A DDC scheme for accessing DF identifiers

As mentioned above, an identifier y in DF can be represented in the DDC by a special FT, shown in Figure 6.4, which accesses H_y, the history structure containing values of $[\![y]\!]$. In the figure, the network labelled *Access* stands for a scheme which accepts a structure input H and an index i, applies the R operator to H i times, and then the L operator, thus accessing the correct value in H. Because H is an ECDS, if an R or L operation is applied to an undefined structure, the operation will be queued until the structure element is defined.

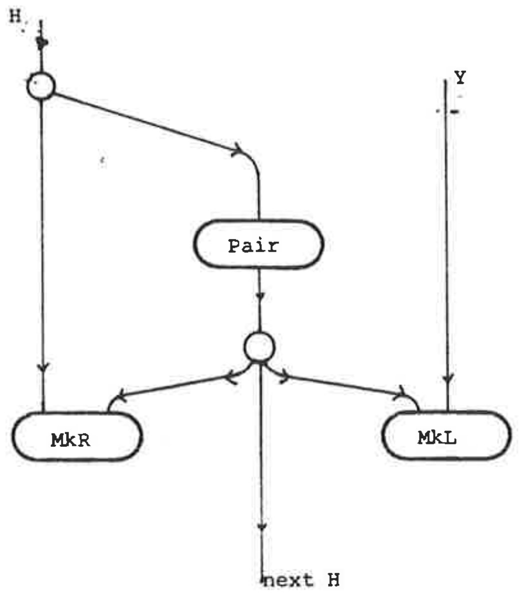


Figure 6.3. A scheme for appending a value to a history structure.

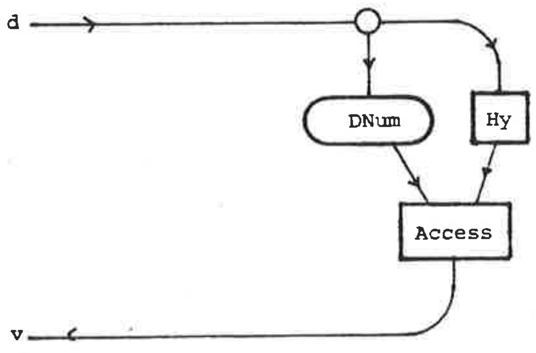


Figure 6.4. A DDC scheme for accessing DF identifiers.

6.4 An example of program translation

Consider the LX3 program shown in Figure 6.5, the translation of which is described in detail in Appendix 3. It can be described operationally as producing a result, formed by first extracting, under a condition specified in an **asa** definition, a value from $[\![b]\!]$, where b is defined in the program using an equation, and secondly finding the factorial of that value.

Suppose that the program is to be generalized in such a way that it computes the factorial of a value, not from $[\![b]\!]$, but from some nonpointwise transformation of $[\![b]\!]$. LX3 is not suitable for expressing such functions, making it necessary to rewrite the program in LX.

```
prog Fact;
   int b, c,Fac,result;
   define Fac (int n) freezing all;
      int result, i, f;
      result=asa i eq n then f easa;
      i= 1 fby i+1;
      f= 1 fby f*next i;
   edefine;
   c=Fac(b);
   b= 1 fby b+1;
   result=asa next b eq 5 then c easa;
eprog
```

Figure 6.5. Factorial program.

However, it would be preferable, for reasons of efficiency, to use some of the data flow graphs produced by the LX3 implementation, particularly those which compute factorials. Using the language extensions proposed in this chapter, the program of Figure 6.6 is suggested, in which the definition of *filter* defines the transformation to be applied to *b*.

```
prog Fact;
   int b, c, Fac, result;
   int filter in dd;
   define Fac (int n) freezing all;
      {as in Figure 6.5}
   edefine;
   define filter (int p);
      {definitions written in LX}
   edefine;
   c=Fac(filter(b));
   b= 1 fby b+1;
   result=asa next b eq 5 then c easa;
eprog
```

Figure 6.6. Extended factorial program.

As it is the purpose of this section to illustrate communication between the DFC and the DDC, the above program is simplified by removing the factorial computation defined by *Fac* (the translation of which is discussed in Appendix 3) to give the program shown in Figure 6.7. In addition, *filter* has been given a specific definition, using a global from DF, and the **asa** definition in the main program has been simplified.

Note that *b* has been included in DD, in order to comply with Rule (5). If *b* were

```
prog Fact;
  int f, result;
  int b in dd,  c in dd, filter in dd;
  define filter (int p) using f;
    int result,  r;
    r = wvr (p mod 2) eq 0 then p ewvr;
    result =  r  +  f;
  edefine;
  c=filter(b);
  b= 1 fby b+1;
  result=asa c >  500 then c easa;
  f= 5 fby f*5;
eprog
```

Figure 6.7. A program in the extended language.

included in DF, propagation of demands from the **wvr_then_ewvr** operator in *filter* might cause an attempt to access a value of $[\![b]\!]$ not yet computed by the DFC.

Rule (5) is now explained further. Suppose a DDC FT has been activated (from the DFC) with demand number n. From the point of view of the FT, the DFC is at iteration n; Rule (5) stipulates that, as demands propagate within the DFC, no demand for a "future" value of a DF identifier may be issued. In other words, ECDS elements beyond n+1 (the **next** value) may not be accessed. If this restriction is not made, deadlock is possible. Suppose b is in DF, and consider

$$c \ = \ filter(b)$$

when the FT for *filter* has been invoked with demand number n and an ECDS H_b representing b. Clearly, if the FT for *filter* generates a demand number of say, n+5, directed to p, the formal parameter of f, an attempt to access $H_b[n+5]$ will occur. This access will be delayed pending computation of $[\![b]\!]$ at iteration n+5; in other words, iteration n cannot complete before iteration n+5, a case of deadlock.

The following histories satisfy the definitions of the program in Figure 6.7:

$$\begin{aligned}
[\![b]\!] &= \langle\ 1,\ 2,\ 3,\ 4,\ \ldots\ \rangle\\
[\![n]\!] &= \langle\ 5,\ 25,\ 125,\ 625,\ \ldots\ \rangle\\
[\![f]\!] &= \langle\ 7,\ 29,\ 131,\ 633,\ \ldots\ \rangle\\
[\![result]\!] &= \langle\ 633,\ 633,\ 633,\ 633,\ \ldots\ \rangle.
\end{aligned}$$

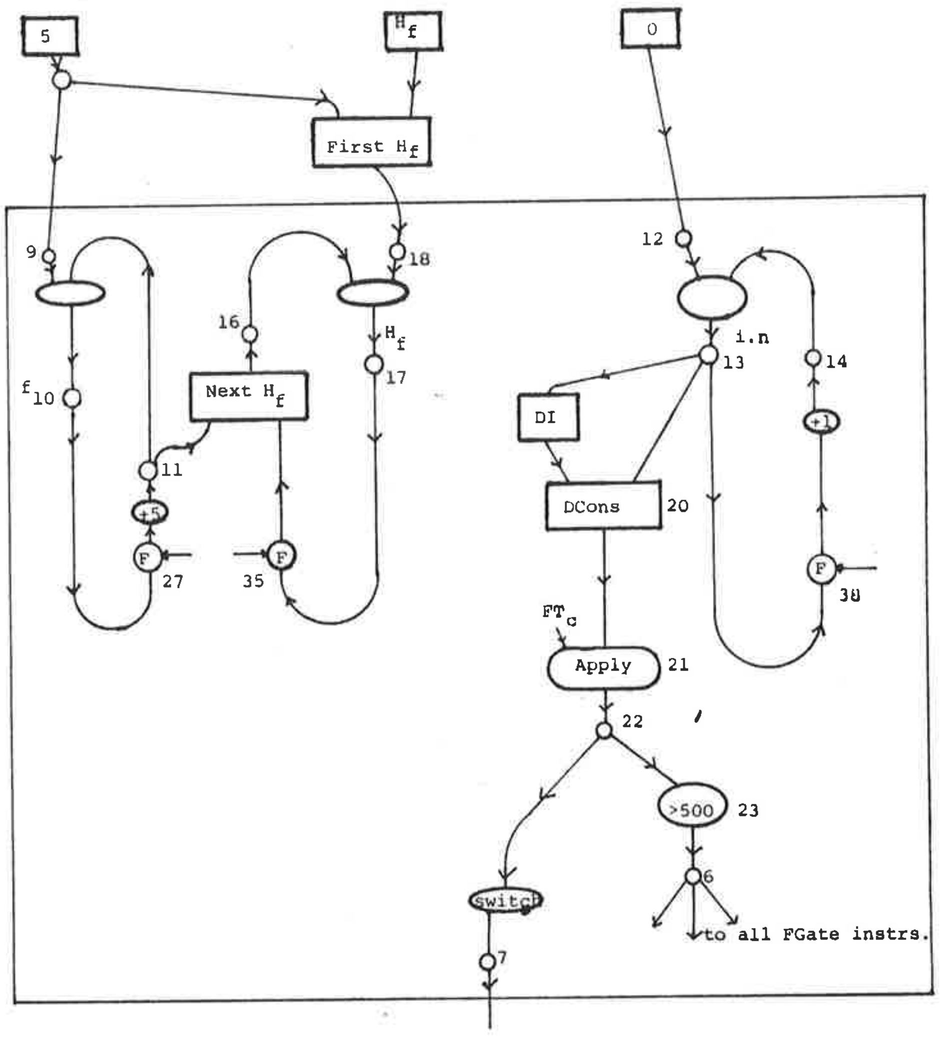


Figure 6.8. DFC of program of Figure 6.7.

Note that $[\![c]\!]$ can be described thus:

$$c_i \;=\; \text{“the } i{+}1^{st} \text{ even number in } [\![b]\!]\text{”} + f_i.$$

The data flow graph of the DFC is shown, in cyclic form, in Figure 6.8. The additional circulators required to construct H_f (which is needed because f is used by *filter*) and to circulate the iteration number are clearly marked. The unexpanded graphs *FirstH*$_f$ and *NextH*$_f$ correspond to *FirstH* and *H++y* described in §6.3.1.2, and *DI* creates a dummy *instance* component for the demand transmitted into the DDC via the FT for *c*.

Execution of the graph of Figure 6.8 is initiated by an external trigger which enables the CONSTANT instructions labelled c_1, c_2 and c_3. It is interesting to note that there are two distinct components of the graph. In the first, the circulators for f and H_f produce values which are appended to the structure H_f. The second component is driven by the circulator for $i.n$ producing successive demands propagated to the DDC which consumes values of H_f and returns values in $[\![c]\!]$ for use in determining the termination condition and result of the DFC.

Note that c would normally be in DF; it is placed in DD in this example to simplify construction of the data flow program. With c in DD, the LX implementation contructs an ET to be passed to *filter*; while this could be done by the DFC, it is more convenient to permit it to be done by the DDC.

Further details of the construction of a data flow program, and its execution on the data flow graph interpreter, are given in Appendix 5.

6.5 Further examples

In this section, two additional examples are given of the development of programs in the extended language. The purpose of these examples is to give an indication of the applicability of the hybrid scheme. The first shows an LX3 program modified, using the extended language, to include a nonpointwise transformation; the modified program is found to violate Rule (5), but a simple reformulation of some definitions is given which solves this problem. The second example shows the extended language used in a simple manner to express additional definitions with much greater clarity than is possible in LX3.

6.5.1 Prime numbers

Considering the program *PRIME* shown in Figure 2.7 of Section 2.3.2.2, it will be recalled that the definitions of the program are satisfied by the following histories; the finite prefixes shown are the values actually computed by the loop corresponding to the program:

$$\begin{aligned} [\![n]\!] &\ \langle\, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, \ldots \,\rangle \\ [\![isprime]\!] & \\ =[\![result]\!] &\ \langle\, t, t, t, f, t, t, f, t, t, f, \ldots \,\rangle \\ [\![stop]\!] &\ \langle\, f, f, f, f, \ldots \,\rangle. \end{aligned}$$

It is desired to modify the program to define *primes*, a sequence of prime numbers, as well as the variable *isprime*; *primes* would then satisfy:

$$[\![primes]\!] \ \langle\, 3, 5, 7, 11, 13, 17, 19, \ldots \,\rangle.$$

Clearly, *primes* cannot be defined as a pointwise transformation of n and *isprime*, that is, $[\![primes]\!]_i$ (abbreviated $primes_i$) cannot be determined either in terms of n_i and $isprime_i$, or in terms of values of those histories in the prefix 0 to i. The values n_i and $isprime_i$ are produced at iteration $i+1$ of the data flow computation; it follows that $primes_i$ cannot be produced at this iteration.

In fact, the variable *primes* cannot be defined directly in LX3. It is, however, possible to define a variable *lastprime* which keeps track of the last value of n which was prime; the definition

$$lastprime \ = \ 3 \ \mathbf{fby\ if\ next}\ isprime\ \mathbf{then\ next}\ n\ \mathbf{else}\ lastprime\ \mathbf{eif}$$

is satisfied by the histories

$$\begin{aligned} [\![n]\!] &\ \langle\, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, \ldots \,\rangle \\ [\![isprime]\!] &\ \langle\, t, t, t, f, t, t, f, t, t, f, \ldots \,\rangle \\ [\![lastprime]\!] &\ \langle\, 3, 5, 7, 7, 11, 13, 13, 17, 19, 19, \ldots \,\rangle. \end{aligned}$$

If, in other parts of the LX3 program, it is required to manipulate a sequence of prime numbers, it is necessary to define such manipulations in terms of both *lastprime* and *isprime*. In summary, it is clumsy to define and use a history of prime numbers in LX3.

Using the extended language of this chapter, it is possible to declare *primes* **in DD**, and define directly

$$primes \ = \ \mathbf{wvr}\ isprime\ \mathbf{then}\ n\ \mathbf{ewvr}.$$

This has two advantages. It is a direct definition of *primes*, and it permits other parts of the program (in either the DFC or the DDC) to manipulate a sequence of prime numbers directly.

It is, however, necessary to use *primes* carefully, to ensure that Rule (5) is satisfied. For example, consider a variant of the original program (Figure 2.7), shown in Figure 6.9, which in fact does violate Rule (5). The "termination condition" of the loop becomes *true* during the tenth iteration; at that iteration, the index is 9, n_9 is 21, and it can be seen that

$$\begin{aligned} stop_j &= result_9 \quad \forall\, j, \text{ and} \\ result_9 &= primes_9. \end{aligned}$$

In other words, the DFC attempts to use $primes_9$. It is the definition of *primes* which violates Rule (5), because the **wvr_then_ewvr** operator, in attempting to compute $primes_9$, causes a demand number greater than 9 to be propagated to *isprime* and *n*, thus attempting to access values not computed by the loop. Consideration is now given to avoiding this violation of Rule (5).

```
prog PRIME;
  int n; bool stop,isprime; int primes in dd;
  n = 3 fby n+2;
  result = primes;
  primes = wvr isprime then n ewvr;
  stop = asa n ≥ 20 then result easa;
  define isprime using n freezing all;
    int i;
    bool idivn,result;
    i = 2 fby i+1;
    result = asa idivn or (i * i ≥ n) then not idivn easa;
    define idivn using n, i freezing all;
      bool result; int m;
      m = 2 * i fby m+i;
      result = asa (m ≥ n) then m eq n easa
    edefine
  edefine
eprog
```

Figure 6.9. A variant of program PRIME which violates Rule (5).

Demands to *primes* must be kept within the finite prefix of $[\![primes]\!]$ actually computed by the loop. From the previous paragraph, it can be seen that the definition of *stop* initiates the demand number propagated to *primes* which causes the problem, suggesting that it may be useful to reformulate this definition. It should be remembered also that in this program, the purpose of the expression component of the **asa** definition is not so much to define a result as to provide a "root" for the dependency graph, and hence it can be altered within this constraint.

The definition is reformulated by once again introducing a mechanism to keep track of those primes computed by the iteration, but using LX instead of LX3. In fact, if *lastprime* is defined as

$$\textit{lastprime} \;=\; \textbf{upon next } \textit{isprime} \textbf{ then } \textit{primes} \textbf{ ewvr},$$

then $[\![\textit{lastprime}]\!]$ is exactly as shown earlier. The definitions of the main program can then be written to satisfy Rule (5):

```
n =  3 fby n+2;
result = primes;
primes = wvr isprime then n ewvr;
lastprime = upon next isprime then primes eupon;
stop = asa n ≥  20 then lastprime easa;.
```

The program no longer violates Rule (5). Consider *lastprime*, which is in DD, and is defined using the DF identifiers *isprime* and *result*; the **upon_then_eupon** operator ensures all demand numbers propagated to *primes* are within the range of values computed by the DFC. From the following history associations

$$\begin{array}{rl}
[\![n]\!] & \langle\, 3,\ 5,\ 7,\ 9,\ 11,\ 13,\ 15,\ 17,\ 19,\ 21,\ \ldots\,\rangle \\
[\![\textit{isprime}]\!] & \langle\, t,\ t,\ t,\ f,\ t,\ t,\ f,\ t,\ t,\ f,\ \ldots\,\rangle \\
[\![\textit{lastprime}]\!] & \langle\, 3,\ 5,\ 7,\ 7,\ 11,\ 13,\ 13,\ 17,\ 19,\ 19,\ \ldots\,\rangle. \\
[\![\textit{primes}]\!] & \langle\, 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ 19,\ \ldots\,\rangle \\
[\![\textit{stop}]\!] & \langle\, 19,\ 19,\ 19,\ 19,\ \ldots\,\rangle
\end{array}$$

it can be seen that

$$\textit{lastprime}_9 = \textit{result}_6 = \textit{primes}_6 = 19.$$

For both the LX3 and extended language versions of the program PRIME, it proved useful to define a variable (*lastprime* in this example) which "padded out" an essentially nonpointwise transformation to match the rate of production of values by the DFC iteration. In this case, there is little apparent difference, in terms of complexity and clarity, between the respective definitions of *lastprime*. It is worth noting that, in many cases, the required nonpointwise transformation will be considerably more complex than that presented here, and much more easily and clearly expressed in LX than LX3.

6.5.2 Averages

Figure 6.10 shows an LX definition which can be used to produce running averages of its argument, which yields a history with the property:

$$[\![avg(z)]\!]_i \quad = \quad \text{average value of } [\![z]\!]_0, [\![z]\!]_1, [\![z]\!]_2, \ldots, [\![z]\!]_i.$$

The definition cannot be written as given in LX3 because it uses several values in the history denoted by its parameter, hence its parameter cannot be frozen.

```
define avg( int x );
   int result, s, n;
   result = s div n;
   s = x fby s + next x;
   n = 1 fby n+1;
edefine.
```

Figure 6.10. A definition of running averages.

Suppose that the factorial program presented in Section 6.4 is to define running averages of factorials. It can be modified by firstly declaring a new variable *avg* **in DD**, defined as above, and secondly declaring *avfac* as a DF identifier, defined thus, assuming that *c* is in DF:

$$avfac \quad = \quad avg(c).$$

The body of the main program can then be written:

```
c = d(b);
b = 1 fby b+1;
avfac = avg(c);
result = asa c> 500 then avfac easa.
```

It is now shown as follows that *avg*(c) does not violate Rule (5). To compute *avg*(c) at some time i, it is necessary to compute *result* in the definition of *avg* at time i, which requires s_i, and in turn c_i; in other words, no value in the future of c is required.

6.6 Discussion

6.6.1 Advantages of the hybrid scheme

The hybrid scheme presented in this chapter facilitates an approach to program development whereby as much programming as possible is done in the relatively efficient LX3, but with parts of the program written in LX when this is necessary or more convenient. When compared with development of a program entirely in LX, it is suggested that use of the techniques outlined in this chapter should give a considerable performance improvement over an unoptimized LX program.

The question arises as to why unoptimized LX should be used at all. Clearly, if the language LX and data flow machines were widely used and well understood, the most satisfactory implementation would be a sophisticated optimizing compiler. However, the state of the art is that developing and debugging LX programs is not well understood, and neither is their implementation on data flow machines. Thus there is a place for translators which use current compiler construction technology as much as possible, and are capable of producing target language code which exhibits clear correspondence with the original source. Both translation methods described in this thesis have these characteristics. The hybrid scheme offers performance improvement with little effect on either of these characteristics; it is, however, emphasized that it is seen as a scheme which is best used in conjunction with further optimizations, separately applied to the LX and LX3 components of a program.

The hybrid translation technique is most obviously applicable to programs of the form suggested in Figure 6.6 of Section 6.4. This program scheme is representative of a class of programs which can be structured into two components: one which produces some history, and another which applies a transformation to that history. The translation technique is directly applicable when the former can be specified in LX3, and the latter component satisfies the restrictions given in Section 6.2. Examination of the (currently limited) repertoire of LX and Lucid programs [AshW76, AshW77b, Fau83, Wen82, Pil83] suggests that this class of programs is quite large.

6.6.2 Early completion data structures

The ECDS plays an important role in the hybrid scheme in embodying the interface between the DFC and the DDC; this interface is effectively an early completion buffer, with the DDC consuming items placed in the buffer by the DFC. The fact that it is an ECDS permits production, consumption and order of access to be seen as independent, asynchronous activities. This results in a clean "factoring" of two important aspects of the scheme: its implementation in a translator, and the run-time behaviour of the target code. In the first case, the translation schemes described in previous chapters can be carried over essentially intact; as shown in earlier sections, the only changes necessary are for communication with the other component, which can be expressed in terms of ECDS operations. Similarly, the run-time behaviour of each component can be considered independently, except when communication between components is required.

The buffering provided by the ECDS is such that a DFC circulator can insert a value in to its history structure as soon as the value is produced, irrespective of consumption by the DDC. If the DDC attempts to retrieve a value from the ECDS before it is computed, the ECDS ensures that the access activity is suspended until a value is written to the appropriate point in the buffer.

The ECDS is used to interface a DFC and a DDC. It would be interesting also to explore the possibility of building a similar interface between a data flow scheme and, for example, a conventional, control flow program. The interface developed in this chapter relies on a common notion of "demand"; construction of a control flow component would similarly depend on appropriate mechanisms for recognizing and generating demands.

6.6.3 Discussion of Rule (5)

Originally, it was hoped that communication via an ECDS would permit the use of arbitrary LX definitions of DD identifiers. It was shown in Section 6.4 that this may lead to deadlock. Hence, Rule (5) was introduced to place restrictions on the consumption of values by the DDC.

The necessity for Rule (5) follows directly from the restrictions which are imposed on LX3 and which guide the construction of the DFC. Consider variables x in DD and y in DF. In the DFC, only 3 components of $[\![y]\!]$ are visible at one time—**first**, **next** and "current". The DDC has access to all components of $[\![x]\!]$, where x is in DD. Consequently, the usual DDC demand mechanism cannot be applied to the representations of DFC variables passed to the DDC, without appropriate restrictions.

Rule (5), as stated, is not entirely satisfactory, and it is probable that further investigation will yield a better formulation of the rule. It would be useful to investigate situations in which the rule could be checked syntactically; Wadge's cycle sum test [Wad81] and Pilgram's "index offset" considerations [Pil83] provide a starting point. It is worth noting that the rule can be checked during execution by tagging the demand, as it passes from the DFC to the DDC, with the current iteration number, and using it to validate all ECDS accesses; improvements to this scheme could also be investigated.

Another aspect for future investigation is to prove that Rule (5) does prevent deadlock. It seems likely that a program satisfying Rule (5) also satisfies Wadge's cycle sum test, which can be applied to some programs to determine whether or not they may deadlock. The test makes use of "index offsets" associated with operators; Rule (5) effectively makes the offsets of definitions of DD identifiers predictable within certain constraints, suggesting that the cycle sum test is satisfied, and providing evidence that Rule (5) prevents deadlock.

CHAPTER 7

DISCUSSION, CONCLUSIONS AND FUTURE WORK

7.1 Introduction

This discussion is concerned with three areas important to the work of the thesis, firstly the design and specification of the language LX is reviewed, and secondly there is an attempt to put in perspective the implementations described in this thesis relative to each other and to other contemporary implementations of Lucid-like languages based in data flow. Thirdly, data flow systems in general are discussed, with emphasis on the expression of demands in data flow.

Various suggestions for further investigation are made throughout this chapter. In particular, the provision of adequate facilities for input/output is seen as an important area for both data flow and LX, and suggestions for future work on this problem are emphasized.

7.2 Language design

Some discussion of language design was presented in §2.2.4, which explored the issues of strong typing and clause structuring. It is apparent that further work is warranted for each of these. For example, considerable declarative information must be included in LX programs, and it can be argued that this represents an overhead which actually reduces the readability of a definition; the relevance of this argument in the context of languages which emphasize conciseness of definition, as LX does, should be investigated.

LX provides basic definitional facilities; investigation of useful extensions to these facilities should be of interest. Some possible additional language constructs follow: a **case** construct for conditional definitions; aggregate operators on histories, to assist in the detection of **forall** parallelism; multiple results from **define** clauses; higher order functions.

The development of data structuring facilities requires considerable attention. Regardless of the details of such facilities, transformations between structures and histories will be important. LX requires that variables be viewed as histories of values; in the case of a history of structure values, it is often useful to manipulate the structure at a particular time index, perhaps generating from it a history of its values which could best be processed iteratively.

LX has only rudimentary features for the expression of input and output. These facilities, and their interaction with aspects of language design such as strong typing, clearly warrant further investigation. In the next section, some suggestions are made for improving these features.

7.2.1 Input/Output in LX

The global variables of an LX program can be regarded as defined externally, with their values supplied as inputs to the program. The history denoted by $[\![\textit{result}]\!]$ in the main program can be interpreted as the output produced by the program. This provides a simple means of getting values to the main program and receiving results from it. Inner clauses may obtain access to external input sources by inheriting global variables; they may contribute to the output of a program by returning results to the outermost (program) level for inclusion in $[\![\textit{result}]\!]$. It is also possible to establish a form of dialogue if the implementation ensures that output is produced as soon as available, and input accepted when supplied.

These facilities are inadequate in many respects. For example, consider how the format of output can be specified. If the main program ***result*** is of type T, the values can be output in a fixed, implementation-determined format appropriate to values of the type; there are no facilities in the language for defining alternative formats. Using stream oriented output within the framework of LX, the output may be considered as a history of characters, and formatting may be specified by including appropriate control characters. Thus, it is necessary to devise a means of deriving, from the history denoted by x of type T, a stream of characters capable of producing the desired output. That is, a history of type T must be transformed into a history of characters. A general input/output facility would need to contain a library of such transformations, which are inherently non-pointwise. Another approach is to specify formatting requirements using a variant of the format descriptions of Fortran, permitting a higher level description from which the implementation could generate the required stream of characters.

Further problems arise if an LX program is to fit into an existing environment. For example, it may be necessary to associate more than one history with an input/output source (or vice versa), to describe the structure of a file, or to display, in some form, a variable defined at an inner level. All are worth further investigation; some attention is now given to the latter.

Confining expression of input/output to the outermost level of an LX program is unnecessarily restrictive. Consider a display facility, whereby any variable may be annotated to indicate that a suitable display of its values is to be output from the program; such a facility is a simple example of an inner level specification of a requirement for output. A proposal, presented in [Wen83], represents an attempt to define the semantics of such a facility in terms of language, rather than implementation, concepts. The essential idea is to introduce program tranformation rules which firstly introduce extra results corresponding to displayed histories, and secondly move the specifications of those histories to the outer level, where they can be associated with output devices. It is also important that a display specification does not initiate computation which is redundant with respect to the main result; [Wen83] outlines how such a display might be defined in LX. These ideas are at an early stage of development, and require further investigation.

7.3 Comparison of implementations

Several proposals for the implementation of Lucid and related languages have been put forward. It is worth examining these developments chronologically, to put in perspective the work reported in this thesis in comparing it with other proposals. The work of this thesis began in 1978, inspired principally by [AshW77a] in conjunction with a desire to find a higher level approach to programming. It was decided to perform an experiment testing the feasibility of implementing "Lucid" using conventional compiler construction techniques, in particular, the recursive descent approach espoused by Wirth [Arv80]. It was also decided to treat Lucid principally as a language for describing iterative computations, a view encouraged by the perspective placed on the language in [AshW77a], hence the implementation was of a subset (Lucid-W [Wen81]) which emphasized these features. This work was carried out shortly after, but independently of, the subset implementations of Hoffmann [Hof80] and Farah [Far77], as discussed in §4.6.4. It came some time after, but was not influenced by, the full Basic Lucid interpreters of Cargill [Car76] and May (mentioned in [AshW77a]).

7.3.1 The implementations of LX3 and LX

The earliest implementation of Lucid-W (which is essentially LX3, apart from some minor syntactic differences) generated imperative code. At this time, suggestions had been made about relationships between Lucid and high level data flow [AshW77a], and there were interesting developments in low level data flow [Den74, Mis77] as a means of exploiting parallelism. In order to gain some insight into relationships between Lucid, a language with a mathematical foundation and roots in the field of program verification, and data flow, a novel basis for new, parallel machine architectures, an experiment was planned to translate LX3 to low level data flow. This implementation is described and discussed in Chapter 4 of this thesis. It was concluded that LX3 could be considered a suitable language for data flow machines, as it translates naturally to standard data flow schemes, and that it is comparable in expressiveness to languages designed specifically for such data flow machines. Chapter 4 also found that LX3 is more naturally implemented on a data flow than a sequential machine, in that both analysis of data dependencies between definitions and code generation are simpler in the data flow case.

Although the results of these experiments with the translation of LX3 were encouraging, it was always an important goal of the research to extend the techniques to translate successively more powerful subsets of Lucid. To assist subsequent discussion of attempts to do this, a simple model is introduced, emphasizing those properties of data flow which are particularly relevant to the implementation of Lucid and its variants. Three important aspects of data flow implementations of Lucid are distinguished: computation agents, regulation, and storage. The first encompasses those objects in a data flow scheme which produce histories. For example, in the translation scheme used in Chapter 4, a primitive operator may be the computational agent for the history denoted by a simple expression, while a circulator is the agent responsible for the production of the history denoted by an inductive variable. Regulation refers to that aspect of a data flow scheme which controls the production of values; in the scheme of Chapter 4, the loop termination condition regulates the number of values produced, and the control operations of a loop regulate the rate of production. Storage refers to storage of history values; in the data flow graphs used in Chapter 4, it is provided by the arcs of the graph.

In the LX3 translation scheme, the computational agents (described above) are simple, and entirely data driven; an agent places a history value on the arc representing the history according to the usual firing rules of data flow operations. Regulation is similarly data driven; all control is expressed in terms of values in histories.

Two key assumptions make this simple regulation scheme possible, and generally simplify the translation of LX3. The first is that variables can be identified with arcs of a data flow graph, and that the values flowing along the arc represent the values in the history denoted by the variable in index order. The second is that histories defined are such that a loop scheme can be used to synchronize the computations, in index order, of every value in each of the histories denoted by several inductive variables. In other words, given some interrelated inductive variables, it is assumed that a circulator is the appropriate computational agent for each, that every value in each history is to be computed, and that the circulators can be synchronized by a common index.

In moving towards the implementation of a larger subset of LX, several attempts (not reported in this thesis) were made to develop translation schemes for nonpointwise operators such as **wvr_then_ewvr**. Clearly, such an operator violates the synchronization assumption, and it is no longer possible to use only loop control for regulation.

At first sight, a promising approach seemed to be the development of a more sophisticated regulation scheme. The suppression of redundant computation was also an important aspect in developing a new regulation scheme. Circulators remained as the basic computational agent for both inductive definitions and those using nonpointwise operators, the first assumption above, concerning the identification of arcs with histories, was retained; it was required, however, that the scheme use only standard data flow operations. Circulators for definitions using nonpointwise operators were seen as asynchronous (in the sense of not sharing a common index) agents separate from those in an LX3 style loop component.

A circulator can be viewed as a computational agent which produces the next value in a history when triggered by a control value directed to its FGATE instructions. In the loop scheme of §4.2, each circulator is advanced by a common control signal, in the knowledge that values required for the computation will be available. However, with circulators viewed as asynchronous agents, no such common signal could be used, so attempts were made to design schemes for circulators which, in effect, used control signals as demands, in that a circulator interpreted an incoming signal as a request, and would send a signal as a demand to those circulators which produced values needed for the requested computation. For example, one idea explored was to express Henderson's **delay** and **force** primitives [Hen80] in terms of data flow control values and instructions. However, a satisfactory regulation scheme which managed demands for required values only, maintained communication between the synchronous and asynchronous components, and still retained the flavour of a data driven network, could not be developed.

As a result, it was decided to change strategy at this stage (mid 1982). It was apparent that quite sophisticated extensions would be required to the basic LX3 scheme to implement even a slightly larger subset of Lucid. Given that this would almost certainly be considerably more expensive than the basic scheme, it was decided to attempt the derivation of translation schemes for an unrestricted language, working directly with demand flow to get a translation scheme in complete agreement with the mathematical semantics of LX. Possible refinements to "more natural" DDF-like schemes are left for further investigation.

It was found necessary to remove the assumption that variables be identified with arcs and arcs with histories, with the attendant implication that values in a history flow along the arc in time order. There are two reasons for this: firstly, with this assumption, it is impossible to provide a complete and correct translation of all programs [AshW77a]. This follows from the fact that all values which flow on an arc must be requested and produced in order of increasing "time"; it is possible to write legal Lucid programs which do not conform to this restriction, for example, the factorial program of Appendix 4. Secondly, intermittent histories (§1.8) cannot always be produced correctly.

In the resulting implementation scheme for LX, described in Chapter 5, the computational agents are function templates, but it is important to note that such an agent is not responsible for the computation of an entire history, but rather for an individual value in that history. Regulation is provided by the DTs, which manipulate demands directly; this provides sufficient flexibility to ensure a correct implementation. Storage is provided by the arcs of the graph; the restrictions of pipelining are avoided by the recomputation of values—a data flow activity is spawned for every value required.

The principal contribution of this scheme is the semantic model, which provides a concise operational description of how demands are propagated. Considered as an implementation, the emphasis on recomputation of individual values is clearly impractical; this problem, and a possible solution, were discussed in §5.4. Because the complete avoidance of redundant computation provided by the LX scheme may well only be essential in a few situations, the hybrid scheme of Chapter 6, by enabling selective use of LX schemes, should be useful also in lessening the amount of recomputation. The hybrid scheme shows also that the contrasting schemes for LX3 and LX can be integrated.

7.3.2 Other implementation schemes

Consideration is now given to the translation schemes of Pilgram [Pil83] and Denbaum [Den83], each of which translates a variant of Lucid to imperative, rather than data flow code. However, in each case the underlying translation schemes can be viewed in terms of data flow in a general sense, and provide some insight into translation to data flow. The development of each of these schemes was contemporary with that of the operational model of LX.

Pilgram's scheme, which is capable of translating almost all full Lucid (the pLucid [Fau83] variant) programs, uses, as an intermediate form, a generalized high level data flow model which retains the assumption that an arc is a pipeline along which flow the values of a Lucid history, in time order. However, this model is based on a graphical view of a Lucid program (§2.2.3), and describes data flow at a much higher level than the model used in this thesis. Pilgram's model is demand driven in that, in addition to values flowing along the arcs, requests may flow in the opposite direction.

Initially, a Lucid program is transformed directly into a graph, in which Lucid operators and user defined functions are nodes, and every arc corresponds to a variable or partial result. This graph is then translated into a system of message passing actors, essentially with one actor for each node, in which the actors use a pre-determined protocol in cooperating with each other to ensure, firstly, that the values transmitted between them are the values of the history associated with the corresponding arc of the original Lucid graph, and secondly, that, whenever possible, values are not computed needlessly.

Consider the mechanisms used to transmit values in Pilgram's scheme. The computational agents are message passing actors; an actor attempts to compute and transmit values one at a time, in index order, and only if needed. Each such actor communicates with others using a standard protocol, under which an actor can receive values and requests. There are three requests: COMPUTE, ADVANCE and NULLIFY. Part of the state of each actor is the history index; COMPUTE requests that the value at the current index be computed, and ADVANCE that the index be incremented by one, perhaps without having computed the value at the current index. NULLIFY is used to cancel computations which have been initiated but turn out not to be required; a full account of its significance is beyond the scope of this discussion. Suppose all values in a particular history are required; the cycle of activity for its actor would be to receive a COMPUTE request, compute the value, send it to the requesting actor, then receive an ADVANCE request to move on to the next value "in the pipeline". By way of comparison, this pattern of activity is specifically built into the design of a circulator; the arrival of a control value at its "merge" operation constitutes a combined COMPUTE/ADVANCE request, with consequent (and unavoidable) computation and transmission of the value. In Pilgram's scheme, storage for values computed but not yet demanded is provided by queues at points in the Lucid graph where an arc is forked to multiple points of use. Queueing is necessary because demands arrive from points of use at varying rates.

Clearly, in Pilgram's scheme the computational agents are comparatively complex entities, and are responsible for the regulatory function of the scheme—they supervise the transmission of demands for values, as well as of the values themselves. Pilgram's regulation protocol permits the translation of nonpointwise operators and general user defined functions, but it cannot translate all programs, for example, those in which a variable is defined in terms of its own future. Also, as Pilgram acknowledges, it occasionally initiates redundant computation of a value, but it does have a mechanism by which such computations can often be abandoned.

This suggests that direct translation of extended LX3 to low level data flow, as discussed above, did not work primarily because the "data flow" must be seen at a considerably higher level. Results equivalent to Pilgram's could be achieved by implementing communicating actors in a low level data flow system with process handling primitives, thereby simulating a higher level of data flow. In fact, facilities have been proposed which should make this possible, for example, Id managers [ArvGP78] and the communicating processes of [CatG80].

Denbaum's thesis [Den83] describes another approach to the implementation of Lucid-like languages. The language considered is ANPL, which is essentially Clause Lucid; ANPL's **define** clause subsumes the **compute** and **mapping** clauses of [AshW78], and its **produce** clause the **produce** and **function** clauses of [AshW78]. The thesis presents an operational semantics of ANPL, based on a technique used by Marlin [Mar80] in describing ACL; in this method separate information structure models are developed for the sequence and data control aspects of the language. The semantic models are used as the basis for a compiler which translates an ANPL program to imperative, coroutine-based code, in the form of an ACL program. Thus, it is not strictly a data flow implementation, but has much of the flavour of one, and provides interesting comparisons with both Pilgram's work and this thesis.

In Denbaum's implementation scheme, there is a computational agent for every variable and use of a clause, responsible for computing values in the corresponding history on request; the agent is either a coroutine or a procedure, and is derived from the ANPL definition. Regulation of computation of history values is specified in a novel fashion. Agents do not communicate directly, but rather request the values required to perform a computation by invoking a special procedure ***retrieve***. If the requested value is already available (see below), ***retrieve*** simply returns it; otherwise, it uses a dependency graph to determine which values of other histories are needed to meet the request, invokes ***retrieve*** to obtain them, and then calls or resumes the appropriate computational agent to actually compute the originally requested value.

The requirement to produce and access values in index order is relaxed by allocating storage, a list of the values used, for each history. This eliminates any need to recompute values, and permits values to be produced and accessed in any order. Storage of all values is in direct contrast with the approach used for LX, which, in its simplest form, requires recomputation of all values. Ideally, a suitable combination of storage and recomputation should be determined by the compiler, depending on the program; this is a significant problem in the implementation of many very high level languages.

Denbaum's scheme is not entirely correct, however, in that it does not always agree with the mathematical semantics. A coroutine is used as the computational agent for an inductive variable; the only requests which can be issued to a coroutine for the production of values are **create**, which establishes the coroutine instance and returns the **first** value, and **resume**, which computes the **next** value. Such an agent cannot correctly implement an intermittent history, for example, one in which the **first** value is not required. Further, it is apparent from the semantics of Lucid that a recursive function invocation effectively introduces a new history; it seems that Denbaum's model does not assign storage for histories introduced in this way, and hence it cannot correctly implement all recursive functions.

Some general conclusions can be drawn about data flow implementations of Lucid and similar languages. The implementation of LX3, and those of Pilgram and Denbaum, are examples of attempts to implement Lucid by adopting a particular operational interpretation of the language; in each case, the particular interpretation cannot be applied successfully to all programs. Although it is desirable, from the point of view of efficiency, to adopt a consistent operational interpretation, it seems difficult to find one such interpretation which encompasses all programs. The operational semantic model of LX is an attempt to do this in terms of demand driven computation; the implementation achieves agreement with the mathematical semantics of LX, but at the expense of recomputation of values and a quite complex regulation scheme. This would suggest that a practical implementation of Lucid should be a hybrid, capable of exploiting different operational interpretations according to the characteristics of particular programs; the implementation techniques described in this thesis and elsewhere are tools which could be incorporated into such an implementation.

7.4 Data flow models

Data flow models were discussed at length in Sections 1.5 and 3.5, particularly the relationship of the model used in this thesis to cyclic schemes, the usability of the graphs produced on data flow machines, and the relevance of early completion data structures. Additional points of interest are presented in this section.

7.4.1 Uses for early completion data structures

As described in Chapter 6, the principal use of early completion data structures for the research reported in this thesis is to implement an incrementally constructed buffer between the data flow and demand driven components of a hybrid translation scheme. However, there are other ways in which they can be used, and these are now discussed briefly.

It was mentioned in §3.4.4 that such structures could be used to pass arguments to functions. In code generated by the LX translator, the only arguments passed to function activations are demands; the facility would permit the components of a demand, particularly the environment table, to be passed incrementally, thereby providing an opportunity for speeding the progress of demands. Values could also be returned incrementally, particularly list structured values, thereby giving a "lazy evaluation" semantics to the lists of the language.

The possibility of associating storage with histories in the demand driven implementation was discussed in §5.4. It is apparent that an early completion data structure should be used to implement such storage, because the order in which components of the history would be produced and accessed is unpredictable; it is necessary to delay access to a particular element pending its computation. Note that use of the history structure is essentially functional; once an element is written to a history, it is never changed.

It is expected that early completion data structures will also be important in implementing a useful input/output facility. This will be discussed further in §7.4.4.

7.4.2 Combining data and demand driven data flow

Attempts to combine demand driven and data driven data flow have been significant in this research. A first approach, not described further here, was to take a data driven graph and superimpose on it additional data flow operations which would cause the operations of the original graph to behave in a demand driven fashion. This is quite straightforward for arithmetic operations. Consider an operation with two inputs and one output. It can be made demand driven by adding an IDENT operation with one input and two outputs, transmitting demands in the opposite direction to the flow of data; execution of the original operation is initiated by a sending an arbitrary value along the additional demand network; operand values are not produced until triggered by a demand.

This simple approach could not be extended to permit demands to be propagated past an IDENT operation, however. The reason is that a demand driven IDENT operation must provide storage for values demanded on some outputs but not on others, as observed by Pilgram [Pil83] (and noted in §7.3). Correct handling of demands requires a demand propagation operation with arbitrary storage requirements and complex internal state transitions; this is very difficult to implement with primitive data flow operations.

This more general approach was rejected principally because, for an adequate implementation, it would have been necessary to go beyond "simple" DDF-like schemes. The approach adopted, and successfully implemented (Chapter 5), is less general in that it uses a notion of demand specific to LX implementation. It is worth noting that, in both cases, the objective was to abstract out that part of a demand driven scheme which handles demands, and express it explicitly in data driven data flow. In other demand driven models, and the hardware which implements them, these demand manipulations are implicit.

Extensions to a data driven data flow model to provide demand manipulation primitives, similarly to the transformation of demands in the operational model of LX, warrant further investigation. These primitives, which could perhaps be implemented in microcode, would provide flexible handling of demands in appropriate situations. This is thought to be particularly relevant for input/output, as discussed in §7.4.4.

7.4.3 The notion of storage in data flow

In pure data flow [DenFL74, ArvG78], there is no notion of assignment to storage; all values, including structures, flow on the arcs and are manipulated by side-effect-free operations—this, of course, is the key to the parallelism provided by the model. In this thesis, the advantages of providing storage for history values have been discussed (§5.4); it is interesting to look briefly at various general notions of storage in data flow.

In a data flow graph, the arcs may be viewed as storage for values; in actual machines, this storage is implemented in various ways, which may be exploited to provide a low-level notion of storage. For example, Todd [Tho81] uses instruction cells in the MIT static data flow machine to implement various storage structures, including arrays. At the graphical level, Wendelborn [Wen82] uses cascaded 3-input MERGE gates to buffer a fixed number of recently computed values in a history. In another early proposal, Kosinki [Kos73] provides a low-level storage cell.

A literal interpretation of functional structures at the machine level is clearly impractical, as it gives rise to the need for extensive copying and movement of data. In an early model proposed by Dennis [Den74], structure values are represented using a heap, with the tokens which flow on arcs carrying pointers to the structures. This is reflected in architecture proposals [Mis78] with a separate unit implementing structure storage and processing; considerable work has been done [Isa79, Ack78] on the design of such a unit, in a way which allows the optimization of structure operations and the sharing of storage.

Various attempts to make functional structure operations more efficient have been mentioned (§1.6). For example, early completion data structures and I-structures permit access to incomplete structures; it is interesting to note that they introduce non-functional operations (§3.4.4) in order to do so. The Manchester machine [WatG82] has a provision for fixed array storage at a node to reduce movement of data (§3.4.4).

Treleaven et al [TreBH82, TreHR82] propose a model which combines a traditional notion of storage with low level data flow (see §1.5). In this model, references to memory locations may flow on arcs, thereby enabling the sharing of memory between instructions; memory may be updated as a consequence of instruction execution. Sharing of memory in this fashion allows efficient implementation of data structures such as arrays.

Other models provide nodes of arbitrary internal complexity; such nodes can implement any storage requirements. For example, in the models of both Pilgram [Pil83] and Faustini [Fau82] nodes are used which contain queues of arbitrary size.

7.4.4 Input/Output in Data Flow Systems

This section discusses input and output in general terms; input is considered as those values admitted to a data flow graph through its input arcs, and output as the values produced at a graph's output arcs. Low level details of interaction with devices, for example, the writing of device drivers in data flow, are not considered. The purpose of the discussion is partly to clarify earlier descriptions of input/output facilities (for example, §3.3.2); it is also intended to demonstrate the usefulness of the demand handling primitives of §7.4.2 in the further development of input/output facilities.

Firstly, "batch" input/output is considered, for queueing, dynamic tagged token and acyclic models, and it is shown that such batch input can be regarded as a form of demand driven input. Secondly, it is argued that demand driven input provides a natural expression of interactive input, and hence provides a unifying framework for both batch and interactive input/output.

7.4.4.1 "Batch" input/output

The term "batch input/output" is used to indicate a system in which all input data is available before execution commences. In a model in which the arcs of data flow graph are regarded as queues (§1.5), input values can be placed on the input arcs, causing the consumer operations of those arcs to fire repeatedly, in the usual manner. Assuming the use of standard schemes which preserve queueing on the arcs, output values will appear in order on the output arcs of the graph. Termination occurs when external input arcs are empty, and there is no activity in the graph. In summary, the input values can be pipelined through the program, and outputs produced in order.

However, generally arcs are not regarded as FIFO queues, because this unnecessarily restricts possible parallelism [ArvGP78, WatG82]. Consider an external input arc connected to some consumer operation in a dynamic tagged token model. In execution, many activities may be generated from this one consumer operation. Values introduced to the data flow system must be tagged with the correct labelling information in order to correctly associate a value with the activity which consumes it. Similarly, multiple activities will be associated with the producer operation of an external output arc, hence outgoing values will be produced in arbitrary order. These values may be arranged in correct sequence either within the program itself, using label manipulation instructions, or permitted to leave the program as they are produced, and sequenced by a separate mechanism, outside the data flow program; in a general-purpose data flow system, the mechanism might be a resource manager [ArvGP78, TreHR82] responsible for the output device.

In the "function template" model used in Chapter 3, special instructions were used to simulate the arrival of a sequence of values at an input point and an output point (§3.3.2). This technique initiates sequentialized activations of the program FT, with each activation processing one input value and producing one output value; this simplified view was adequate for testing all programs used in the thesis in a sequential environment. It can be generalized to simulate, in effect, the behaviour of the dynamic tagged token model, with similar sequencing characteristics.

In an acyclic model, it is natural to view the program as an activation of a function template which accepts an input structure and produces an output structure. Assuming that the structures are early completion data structures, the program can be seen as embedded in a system which includes a process [CatG80, TreHR82] which appends values to the input structure, and another which consumes values from the output structure.

In all the above cases, a real implementation will introduce physical limitations, for example, input devices will require that input be buffered. In each case, the assumption must be made that there are signals which pass to an input handler, indicating that the buffer is either full or has available space. Such signals can be regarded as a form of demand for input. In the case of batch input, such demands can be handled entirely by an interface which regulates the rate of flow of values into the program, and need not be explicitly programmed.

The above data driven "batch" schemes have one distinct advantage: data can be transmitted to the program at a rate determined only by the capacity of the program to accept and process it. A disadvantage is that it is occasionally necessary to supply dummy input values to trigger computations which do not actually use the values; an example is a program which consists of a conditional scheme in which both arms of the condition are input arcs.

7.4.4.2 Interactive input/output

In an interactive data driven program, input is related to previous output from the program, acting as prompts. Conceptually, the output feeds back to the input source, thereby controlling it to some degree. Input is thus controlled by two sources of signals, namely the regulating signals mentioned above, and additional signals dependent on program output.

A program prompt emanating from a particular point inside the program at which input is needed in fact constitutes a demand for input. In a data driven system, the prompting signal must follow a path from that point via an output port of the program to the source of input. Although this is quite feasible within the data driven framework if input and output are incremental structures, it is somewhat unnatural. In summary, input in general can be seen as driven by demands, with the difference between batch and interactive input characterized by the nature of the demand.

Now consider the passage of demands in the demand driven system implemented in Chapter 5. A demand propagates from an output point through the program and thence externally to sources of input, requesting those values required by the computation. In other words, all computation takes place in response to a demand for output, including demands propagated to sources of input. This is similar to interactive input as described above, with the difference that in the interactive case the demand is originated by the program, rather than outside it. Thus, input is demand driven, in some sense, in each case considered.

Clearly, a system driven entirely by demands for output has the advantages that the order of demands determines order of response, and no unnecessary input is requested. The principal disadvantage is that the rate of acceptance of data by a program will be slowed, because input is not accepted until demanded; given that speed of computation is of great importance in many applications of data flow, such degradation is unacceptable.

It is, therefore, suggested that the following design of an input facility warrants further investigation. Initially, determine the nature of the signals required to communicate with input devices. Implement batch input using an interface which generates such signals according to the state of the input buffer, thus allowing points of input which operate in a purely data driven fashion. Then, implement demand driven primitives of the type discussed in §7.4.2. This would enable interactive input to be implemented as above, but allows construction of a direct demand path to the site of input, rather than a roundabout feedback mechanism. It would also permit tailormade demand driven regimes. For example, parts of a program could be made entirely demand driven by accepting demands at certain output points, and propagating them through the program; the operational model of LX, or a variant of it, could be used in designing appropriate demand transformations.

7.5 Conclusion

A goal of the research reported in this thesis was to investigate the practicality of implementing a Lucid-like language in a data flow environment. To this end, the language LX was developed.

The major achievements of the work are a demand driven operational semantic model of LX which shows substantial agreement with the mathematical semantics, the completion of an implementation of LX based on the model, the specification of a subset of LX with expressive power comparable to languages specifically designed for data flow, and the satisfactory implementation of that language. The latter experiment demonstrates the practicality of implementing a subset of LX in data flow. While a truly practical implementation of LX itself has not emerged, several suggestions have been put forward in the thesis for its further development towards that goal, including concrete proposals for a hybrid data and demand driven implementation. Experience thus far with implementing LX illustrates the usefulness of a concise operational description of a language as a starting point in the development of a satisfactory implementation.

Another significant achievement is the use of data driven data flow to express demand driven computation. While the notion of demand used is problem-specific, it is suggested that development of more general demand transforming primitives will considerably enhance data driven systems, particularly for expressing input and output.

APPENDIX 1

OPERATOR DEFINITIONS

The operators **wvr_then_ewvr** and **upon_then_eupon** can be defined formally as

$$\begin{aligned}\textbf{wvr}\ \mathrm{b}\ \textbf{then}\ \mathrm{a}\ \textbf{ewvr} &= \mathrm{wvr(a,b)}\\ \textbf{upon}\ \mathrm{b}\ \textbf{then}\ \mathrm{a}\ \textbf{eupon} &= \mathrm{upon(a,b)}\end{aligned}$$

where *wvr* and *upon* are defined using the following recursive **define** clauses:

```
define wvr(bool a, int b);
   result = if  first a
                then b fby wvr(next a, next b)
                else wvr(next a, next b)
            eif
edefine
define upon(bool a, int b);
   result = b fby if  first a
                      then upon(next x, next y)
                      else upon(x, next y)
                  eif
edefine
```

Note that these definitions assume that *b* is of type **int**; similar definitions can be made when *b* is of another type.

APPENDIX 2

OPERATION OF THE DATA FLOW INTERPRETER

In this Appendix, the operation of the data flow graph interpreter is illustrated by giving two examples of the execution of programs. The first example is a recursive factorial program, and the second shows the production and consumption of a partially defined structure.

A2.1 Recursive factorial

The program text is shown in Figure A2.1, and the corresponding data flow graphs in Figure A2.2. The program consists of two functions; the first is called the "program driver", and is used to invoke the second function, which computes factorials. Before discussing the program in detail, some notational conventions are mentioned. The character "=" preceding a literal value in the program text indicates a constant input to an instruction. The null link 0.0 is used to show that no arc terminates at or emanates from the input or output position concerned. In the case of an output link, this means that any value produced by the instruction for transmission from that point will be discarded.

It has been found convenient to structure programs submitted to the interpreter with a more or less standard program driver for the initial activation. In this program, the program driver, designed for testing purposes, provides a mechanism by which the factorial function can be repeatedly invoked, accepting input from, and displaying results at, a terminal. Instruction 3 of the program driver in Figure A2.1, and in Figure A2.2a, shows the factorial function, the FT with a Uid of 1, invoked with a value requested by the START instruction. Repetition of calls to the factorial function is achieved by recursive application of FT 0 (instruction 5). Because no recursion termination condition is given, the RETURN instruction will never be executed; the operation of the interpreter must be terminated by external intervention.

```
*0                                   Program driver
0  Ident     0.0      :  2.1
1  Return    5.1      :
2  Start     0.1      :  3.2
3  Apply     =1  2.1  :  4.1
4  Ident     3.1      :  6.1 5.2
5  Apply     =0  4.2  :  1.1
6  Fin       4.1      :
7  End       0
*1                                   Recursive factorial
0  Ident     0.0      :  2.2 4.2
1  Return    5.1      :
2  Eq        =0  0.1  :  10.1
3  Switch    10.1 =0  :   5.2 0.0
4  Switch    10.2 0.2 :   0.0 6.1
5  Con       =1  3.1  :  1.1
6  Ident     4.2      :  9.1 7.1
7  Sub       6.2 =1   :  8.2
8  Apply     =1  7.1  :  9.2
9  Times     6.1 8.1  :  1.1
10 Ident     2.1      :  3.1 4.1
11 End       0
```

Figure A2.1. Recursive factorial program

Consider the graph of the factorial function, shown in Figure A2.2(b). An incoming value *n* takes one of two paths through the graph; the path taken when *n* is zero is now considered in detail. Node 0 is enabled by the function activation mechanism; on firing, it propagates *n* to nodes 2, 3 and 4 of the graph. Node 2, comparing *n* against 0, then fires to produce the boolean value *true*, which is transmitted, via the IDENT function of node 10, to the SWITCH gates, nodes 3 and 4. These nodes are then enabled, and can fire simultaneously. In each case, the data input *n* is transmitted via the output labelled *T*. This output link is null for node 4, hence *n* is absorbed when node 4 fires, and travels no further on that path. The firing of node 3 causes the CONSTANT node, 5, to be triggered, and the value 1 to be transmitted, via the MERGE node, to the RETURN node; the activation thus terminates, producing the result, 1. The MERGE node is included in the graph to indicate that values are merged from different paths; in this example, a value arrives on one or the other path, but not both, making the merge operation deterministic. No MERGE instruction is included in the textual form of the program; the same effect is achieved by directing two arcs to the same input link.

If *n* exceeds zero, then after node 4 transmits *n* to the IDENT node 6, the right hand path through the graph is followed. Node 8 represents recursive application of the factorial function (the FT of which is 1), with argument $n-1$; the result returned is multiplied by n (node 9), and returned as the result of the function.

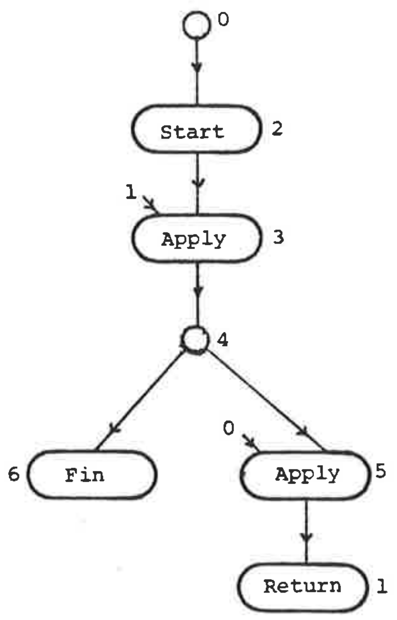
Figure A2.3 shows an extract from a trace of the execution of the program of Figure A2.1, produced by the interpreter. Line 1 shows a request, and user response, indicating that tracing is required for this run. The next line shows that the first function read in has been inserted in the initial heap as the heap node with Uid 0, and that the node is of type *Fn*, namely a function template. Similar considerations apply to the second function read.

Subsequent lines show a user request for the activation of heap node 0 (the program driver), the allocation of an activation on the heap (node 2), and the establishment of instruction 0 as the initial activity. Instruction 0 is an IDENT instruction which, on firing, causes the START instruction to become enabled.

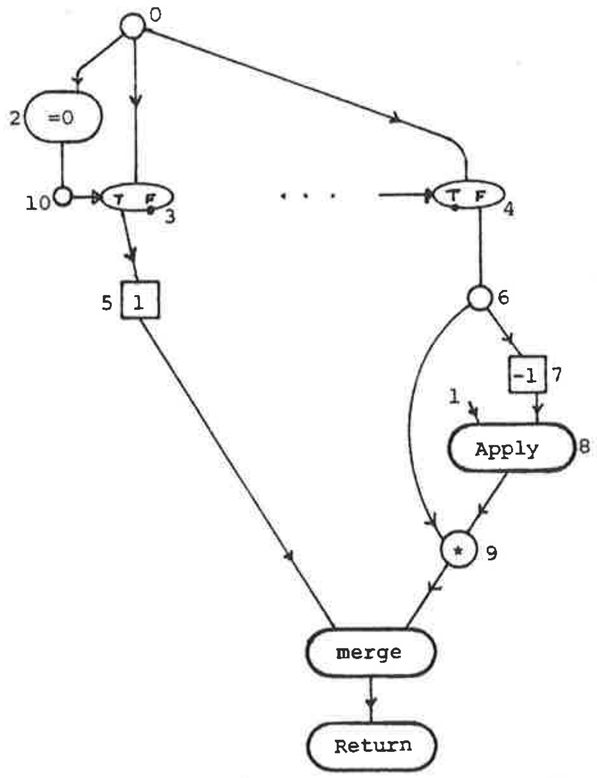
Consider the execution trace for the START instruction, activity [2,2]. The total number of inputs is given by *In*; in this case its value is 1. The value of each constant input, and the position at which it occurs, are listed after *Con*; in this case, there are no constant inputs, hence *InC*, the number of operands which must arrive to enable the instruction, is 0. There is one output, *Out*, directed to destination link 3.2. The input value *Inputs* which triggered the instruction is 0, but this is supplanted by a value requested from the terminal; the response, 3, is the output of the instruction, indicated by *Outputs*.

The trace of the APPLY activity [2,3] (instruction 3 of the activation with Uid 2) shows the information passed to a new activation. The argument value 3 is placed in input position 1 of instruction 0, an IDENT instruction, to create the first activity, [3,0], of the new activation. Instruction 1 is, by convention, the RETURN instruction of the function; identification of the invoking APPLY activity [2,3] is placed in its second input position, and used by the RETURN activity to determine the destination of the result of the function. The notation *NU* indicates that the APPLY activity does not use its outputs; the RETURN instruction, in effect, performs the output transmission phase of the APPLY instruction.

The remainder of the trace follows the course of execution discussed above. Only the APPLY and RETURN instructions of the recursive activation are shown; it can be seen that the value 2 will be returned for factorial 2, and multiplied by the value of n, 3, to give the result 3!. The ellipsis at the end indicates that execution may continue with further factorial computations.



(a) Program driver.



(b) Factorial function.

Figure A2.2. Data flow graphs of recursive factorial program.

```
Data Flow Graph Interpreter
Do you want tracing? y
Heap node 0 allocated as Fn
Heap node 1 allocated as Fn
Which Fn to run ? 0
Activating Fn with Uid: 0
Heap node 2 allocated as Ac
 ACTIVITY LIST
 [2,0]
INTERPRETING ACTIVITY [2,0]
EXECUTING INSTRUCTION
 Ident      In :1 InC:1   Con:
                          Out:1 2.2
            Inputs 0
            Outputs 0
 ACTIVITY LIST
 [2,2]
INTERPRETING ACTIVITY [2,2]
EXECUTING INSTRUCTION
          Start      In :1 InC:0   Con: .1=0
                          Out:1 3.2
            Inputs 0
New value for input 1: 3
            Outputs 3
 ACTIVITY LIST
 [2,3]
INTERPRETING ACTIVITY [2,3]
EXECUTING INSTRUCTION
 Apply      In :2 InC:1   Con: .1=1
                          Out:1 4.1
            Inputs 1 3
Activating Fn with Uid: 1
Heap node 3 allocated as Ac
Opnd 1 of Inst. 0 in Uid 3 = 3
Opnd 2 of Inst. 1 in Uid 3 = [2,3].0
            Outputs NU
 ACTIVITY LIST
 [3,0]
INTERPRETING ACTIVITY [3,0]
EXECUTING INSTRUCTION
 Ident      In :1 InC:1   Con:
                          Out:2 2.2 4.2
            Inputs 3
            Outputs 3 3
 ACTIVITY LIST
 [3,2]
INTERPRETING ACTIVITY [3,2]
EXECUTING INSTRUCTION
 Eq         In :2 InC:1   Con: .1=0
                          Out:1 10.1
            Inputs 0 3
            Outputs F
 ACTIVITY LIST
 [3,10]
INTERPRETING ACTIVITY [3,10]
EXECUTING INSTRUCTION
 Ident      In :1 InC:1   Con:
                          Out:2 3.1 4.1
            Inputs F
            Outputs F F
 ACTIVITY LIST
 [3,4]  [3,3]
INTERPRETING ACTIVITY [3,3]
EXECUTING INSTRUCTION
 Switch     In :2 InC:1   Con: .2=0
                          Out:1 5.2
            Inputs F 0
            Outputs NU
 ACTIVITY LIST
 [3,4]
INTERPRETING ACTIVITY [3,4]
EXECUTING INSTRUCTION
 Switch     In :2 InC:2   Con:
                          Out:2 0.0 6.1
```

```
              Inputs F 3
              Outputs NU 3
 ACTIVITY LIST
 [3,6]
INTERPRETING ACTIVITY [3,6]
EXECUTING INSTRUCTION
 Ident        In :1 InC:1    Con:
                             Out:2 9.1 7.1
              Inputs 3
              Outputs 3 3
 ACTIVITY LIST
 [3,7]
INTERPRETING ACTIVITY [3,7]
EXECUTING INSTRUCTION
 Sub          In :2 InC:1    Con: .2=1
                             Out:1 8.2
              Inputs 3 1
              Outputs 2
 ACTIVITY LIST
 [3,8]
INTERPRETING ACTIVITY [3,8]
EXECUTING INSTRUCTION
 Apply        In :2 InC:1    Con: .1=1
                             Out:1 9.2
              Inputs 1 2
Activating Fn with Uid: 1
Heap node 4 allocated as Ac
Opnd 1 of Inst. 0 in Uid 4 = 2
Opnd 2 of Inst. 1 in Uid 4 = [3,8].0
              Outputs NU
 ACTIVITY LIST
 [4,0]
INTERPRETING ACTIVITY [4,0]
         :
INTERPRETING ACTIVITY [4,1]
EXECUTING INSTRUCTION
 Return       In :1 InC:1    Con:
                             Out:0
              Inputs 2
Return via [3,8]
              Outputs
 ACTIVITY LIST
 [3,9]
INTERPRETING ACTIVITY [3,9]
EXECUTING INSTRUCTION
 Times        In :2 InC:2    Con:
                             Out:1 1.1
              Inputs 3 2
              Outputs 6
 ACTIVITY LIST
 [3,1]
INTERPRETING ACTIVITY [3,1]
EXECUTING INSTRUCTION
 Return       In :1 InC:1    Con:
                             Out:0
              Inputs 6
Return via [2,3]
              Outputs
 ACTIVITY LIST
 [2,4]
INTERPRETING ACTIVITY [2,4]
EXECUTING INSTRUCTION
 Ident        In :1 InC:1    Con:
                             Out:2 6.1 5.2
              Inputs 6
              Outputs 6 6
 ACTIVITY LIST
 [2,5]  [2,6]
INTERPRETING ACTIVITY [2,6]
EXECUTING INSTRUCTION
 Fin          In :1 InC:1    Con:
                             Out:0
              Inputs 6
Result at output 1 = 6
```

Figure A2.3. Execution trace of factorial program

A2.2 Structure production

This example illustrates the use of the structure construction and selection operations. The program includes three functions, namely the program driver, the structure producer, a function which creates and returns a structure value, and the structure consumer, which applies selection operations to the structure.

An important characteristic of the program is that, before the producer has completed its execution, the partially defined structure value is returned, and the consumer activated—this enables the producer and consumer to operate on the structure in parallel. Indeed, the structure value is never completely defined by the producer, but the consumer successfully accesses those elements which are defined.

Figure A2.4 shows the final structure produced by the program, both in tree form, and as represented on the heap. The selection operation R returns the r component of a tree node, hence, given that t denotes the tree depicted in Figure A2.4(a), the following equation holds:

$$r(r(r(t))) \quad = \quad 999.$$

The notation "?" is used in Figure A2.4(a) for an undefined component, represented in a heap node by an empty queue, denoted ***E.Q.*** in Figure A2.4(b). As the interpreter identifies a structure value by its Uid; the notation *STR6* denotes the structure value stored at node 6 on the heap.

The text of the program is shown in Figure A2.5, and data flow graphs in Figure A2.6. Various stages in the construction of the structure are now discussed. Two such intermediate stages are depicted in Figure A2.7, and a trace in Figure A2.8.

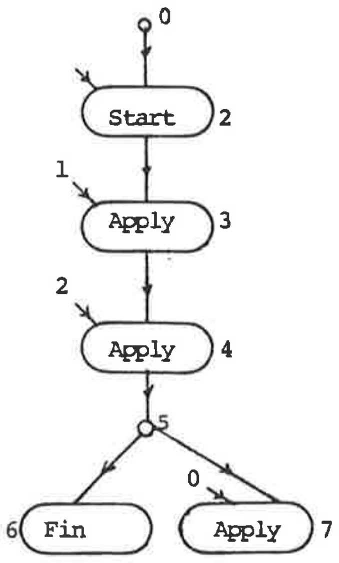
```
       .                     heap node      l       r
      / \                    ---------      -       -
     ?   +                       7        E.Q.    STR6
        / \                      6        E.Q.    STR5
       ?   +                     5        E.Q.     999
          / \
         ?  999
   (a) as a tree             (b) as represented on the heap
```

Figure A2.4. A structure value.

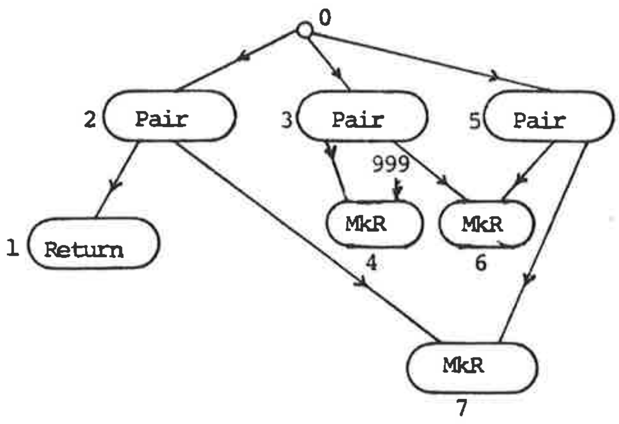
```
0  Ident      0.0          :  2.1
1  Return     0.0          :
2  Start      0.1          :  3.2
3  Apply      =1 2.1       :  4.2
4  Apply      =2 3.1       :  5.1
5  Ident      4.1          :  6.1
6  Fin        5.1          :
7  Apply      =0 5.2       :
8  End        0
*1 production of partially defined structure
0  Ident      0.0          :  2.1 3.1 5.1
1  Return     2.1          :
2  Pair       0.1          :  7.1 1.1
3  Pair       0.2          :  4.1 6.2
4  MkR        3.1 =999     :
5  Pair       0.3          :  6.1 7.2
6  MkR        5.1 3.2      :
7  MkR        2.2 5.2      :
8  End        0
*2 consumption of partially defined structure
0  Ident      0.0          :  2.1
1  Return     4.1          :
2  R          0.1          :  3.1
3  R          2.1          :  4.1
4  R          3.1          :  1.1
5  End        0
```

Figure A2.5. Structure production program

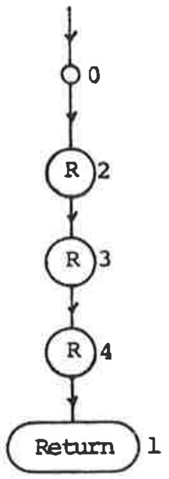
Consider the graph of the structure producer, Figure A2.6(b). Its input parameter acts only as a trigger to enable the PAIR operations 2, 3 and 5, which fire in any order, producing three undefined structure nodes. Next, the construction operations 4, 6 and 7 are then enabled, and can fire in any order to produce the tree representation shown in Figure A2.4(b). Note that the node numbers used in Figure A2.6(b) correspond to those in Figure A2.8, for a particular order of firing of operations 2, 3 and 5. A different order would produce different node numbers, but the nodes would still represent the tree of Figure A2.4(a).



(a) Main program.



(b) Structure producer.



(c) Structure consumer.

Figure A2.5. Structure production program.

The graph of Figure A2.6(b) shows that the root node of the structure may be returned, via the RETURN instruction at node 1 of the graph, immediately after it has been created by instruction 2. It is then used as an argument to the consumer function shown in Figure A2.6(c). The operation of the consumer function is straightforward, merely applying the selection operation R three times (sequentially), and returning the resulting value.

Figure A2.8 shows a trace, produced by the interpreter, of the program in execution. Various points of interest marked on the trace are now discussed.

At point 1, the PAIR activities [4,3] and [4,5] have executed, yielding structure heap nodes 5 and 6; the tree is at the stage of development shown in Figure A2.7(a). Between points 1 and 2, the MKR operations 4 and 6 have fired, and the third structure node has been produced, to give the partially defined tree of Figure A2.7(b); neither component of the root node, 7, has been defined.

At point 3, the root node, still undefined, has been returned from the producer, and has triggered the invocation of the consumer, as activation 8. It can be seen, at point 4, that activity [8,2] attempts to access component R of the root node; because that component is undefined, the activity is placed on the queue associated with the component. The MKR activity [4,2] executes at point 5; this completes the structure to the stage shown in Figure A2.4(b), and also permits the deferred activity [8,2] to proceed. The latter action causes the queue entry to be removed (*DelQ* on the trace), and the *r* component thus selected (*STR*5) to be transmitted to its destination, enabling operation 3 of the consumer. The remainder of the trace shows further elements of the tree successfully accessed, and the value 999 returned as the result of the activation.

```
heap node       l       r
---------       -       -
    6          E.Q.    E.Q.
    5          E.Q.    E.Q.
           (a) at point 1
heap node       l       r
---------       -       -
    7          E.Q.    E.Q.
    6          E.Q.    STR5
    5          E.Q.     999
           (b) at point 2
```

Figure A2.7. Stages in structure construction.

```
Data Flow Graph Interpreter
Do you want tracing? y
Heap node 0 allocated as Fn
Heap node 1 allocated as Fn
Heap node 2 allocated as Fn
Which Fn to run ? 0
Activating Fn with Uid: 0
Heap node 3 allocated as Ac
ACTIVITY LIST
 [3,0]
INTERPRETING ACTIVITY [3,0]
EXECUTING INSTRUCTION
 Ident       In :1 InC:1     Con:
                             Out:1 2.1
             Inputs 0
             Outputs 0
 ACTIVITY LIST
 [3,2]
INTERPRETING ACTIVITY [3,2]
EXECUTING INSTRUCTION
 Start       In :1 InC:1     Con:
                             Out:1 3.2
             Inputs 0
New value for input 1: 77
             Outputs 77
 ACTIVITY LIST
 [3,3]
INTERPRETING ACTIVITY [3,3]
EXECUTING INSTRUCTION
 Apply       In :2 InC:1     Con: .1=1
                             Out:1 4.2
             Inputs 1 77
Activating Fn with Uid: 1
Heap node 4 allocated as Ac
Opnd 1 of Inst. 0 in Uid 4 = 77
Opnd 2 of Inst. 1 in Uid 4 = [3,3]
             Outputs NU
 ACTIVITY LIST
 [4,0]
INTERPRETING ACTIVITY [4,0]
EXECUTING INSTRUCTION
 Ident       In :1 InC:1     Con:
                             Out:3 2.1 3.1 5.1
             Inputs 77
             Outputs 77 77 77
 ACTIVITY LIST
 [4,5]  [4,3]  [4,2]
INTERPRETING ACTIVITY [4,5]
EXECUTING INSTRUCTION
 Pair        In :1 InC:1     Con:
                             Out:2 6.1 7.2
             Inputs 77
Heap node 5 allocated as St
             Outputs STR[5] STR[5]
 ACTIVITY LIST
 [4,3]  [4,2]
INTERPRETING ACTIVITY [4,3]
EXECUTING INSTRUCTION
 Pair        In :1 InC:1   Con:
                           Out:2 4.1 6.2
             Inputs 77
Heap node 6 allocated as St                             (1)
             Outputs STR[6] STR[6]
 ACTIVITY LIST
 [4,6]  [4,4]  [4,2]
INTERPRETING ACTIVITY [4,6]
EXECUTING INSTRUCTION
 MkR         In :2 InC:2   Con:
                           Out:0
             Inputs STR[5] STR[6]
             Outputs
 ACTIVITY LIST
 [4,4]  [4,2]
INTERPRETING ACTIVITY [4,4]
EXECUTING INSTRUCTION
 MkR         In :2 InC:1   Con: .2=999
                           Out:0
             Inputs STR[6] 999
             Outputs
 ACTIVITY LIST
 [4,2]
INTERPRETING ACTIVITY [4,2]
EXECUTING INSTRUCTION
 Pair        In :1 InC:1   Con:
                           Out:2 7.1 1.1
             Inputs 77
Heap node 7 allocated as St                             (2)
             Outputs STR[7] STR[7]
 ACTIVITY LIST
 [4,1]  [4,7]
INTERPRETING ACTIVITY [4,1]
EXECUTING INSTRUCTION
 Return      In :1 InC:1   Con:
                           Out:0
             Inputs STR[7]
Return via [3,3]
             Outputs
 ACTIVITY LIST
 [3,4]  [4,7]
INTERPRETING ACTIVITY [3,4]
EXECUTING INSTRUCTION
 Apply       In :2 InC:1   Con: .1=2
                           Out:1 5.1
             Inputs 2 STR[7]
Activating Fn with Uid: 2
Heap node 8 allocated as Ac
Opnd 1 of Inst. 0 in Uid 8 = STR[7]
Opnd 2 of Inst. 1 in Uid 8 = [3,4]
             Outputs NU
 ACTIVITY LIST                                          (3)
 [8,0]  [4,7]
INTERPRETING ACTIVITY [8,0]
EXECUTING INSTRUCTION
 Ident       In :1 InC:1   Con:
                           Out:1 2.1
             Inputs STR[7]
             Outputs STR[7]
 ACTIVITY LIST
 [8,2]  [4,7]
INTERPRETING ACTIVITY [8,2]
EXECUTING INSTRUCTION
 R           In :1 InC:1   Con:
                           Out:1 3.1
             Inputs STR[7]
InsQ:[8,2]
             Outputs NU
 ACTIVITY LIST                                          (4)
 [4,7]
```

```
INTERPRETING ACTIVITY [4,7]
EXECUTING INSTRUCTION
 MkR        In :2 InC:2   Con:
                          Out:0
            Inputs STR[7] STR[5]
 DelQ
            Outputs NU
 ACTIVITY LIST                                        (5)
 [8,3]
INTERPRETING ACTIVITY [8,3]
EXECUTING INSTRUCTION
 R          In :1 InC:1   Con:
                          Out:1 4.1
            Inputs STR[5]
            Outputs STR[6]
 ACTIVITY LIST
 [8,4]
INTERPRETING ACTIVITY [8,4]
EXECUTING INSTRUCTION
 R          In :1 InC:1   Con:
                          Out:1 1.1
            Inputs STR[6]
            Outputs 999
 ACTIVITY LIST
 [8,1]
INTERPRETING ACTIVITY [8,1]
EXECUTING INSTRUCTION
 Return     In :1 InC:1   Con:
                          Out:0
            Inputs 999
Return via [3,4]
            Outputs
        .
        .
        .
```

Figure A2.8. Execution trace of structure production program.

APPENDIX 3

AN EXAMPLE OF LX3 PROGRAM TRANSLATION

In this Appendix, the operation of the various components of the LX3 data flow implementation is illustrated by examining significant stages in the translation of a simple factorial program, shown in Figure A3.1.

```
prog Fact;
   int b, c,Fac,result;
   define Fac (int n) freezing all;
      int result, i, f;
      result=asa i eq n then f easa;
      i= 1 fby i+1;
      f= 1 fby f*next i;
   edefine;
   c=Fac(b);
   b= 1 fby b+1;
   result=asa next b eq 5 then c easa;
eprog
```

Figure A3.1. Factorial program.

Five stages in the translation of the program are considered below. In addition, the iterative extensions to the data flow model introduced in Chapter 3 are illustrated by briefly considering the execution of the target language program.

(1) Dependency graphs

The main program and the **define** clause are analyzed independently to produce two separate dependency graphs, as described in Section 4.3.2. The graphs are shown in Figure A3.2.

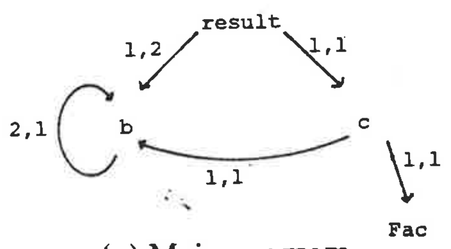
(2) Initial data flow graphs

Figure A3.3 shows the incomplete data flow graphs generated by the source analyzer for the **define** clause and for the main program. The graphs are incomplete in that circulators are formed only partially, and CONSTANT instructions are not triggered. A separate function template is used for each clause and the main program. In each case, as described in Section 3.4, instruction 0 is the initial activity and is used to transmit the first parameter of the **define** clause, and instruction 1 is the

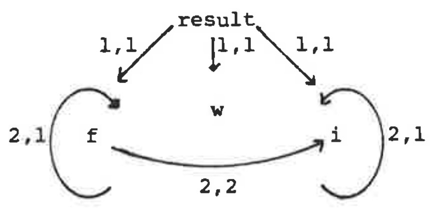
RETURN instruction. The implementation assumes that each identifier requires three characteristic addresses, and hence generates three IDENT instructions when an identifier is declared; as can be seen from the graph produced (Figure A3.3), all of these instructions may not be actually used.

(3) The code template

The dependency analyzer produces a code template as shown in Figure A3.4(b) and A3.4(c). Figure A3.4(a) shows the format of a typical code template entry, and of its sub-lists. At most one of ***gatelist*** and ***occlist*** are used, ***gatelist*** if ***identifier*** is an inductive variable, ***occlist*** if it is quiescent, and neither otherwise. The ***arc*** component of a ***gatelist*** entry records information from the relevant dependency graph arc; for example, the ***gatelist*** for the inductive variable ***f*** includes entries for the two arcs emanating from ***f*** in the dependency graph. The ***gating flag*** component has value ***g*** if a gate is to be generated for each occurrence, ***ng*** otherwise.

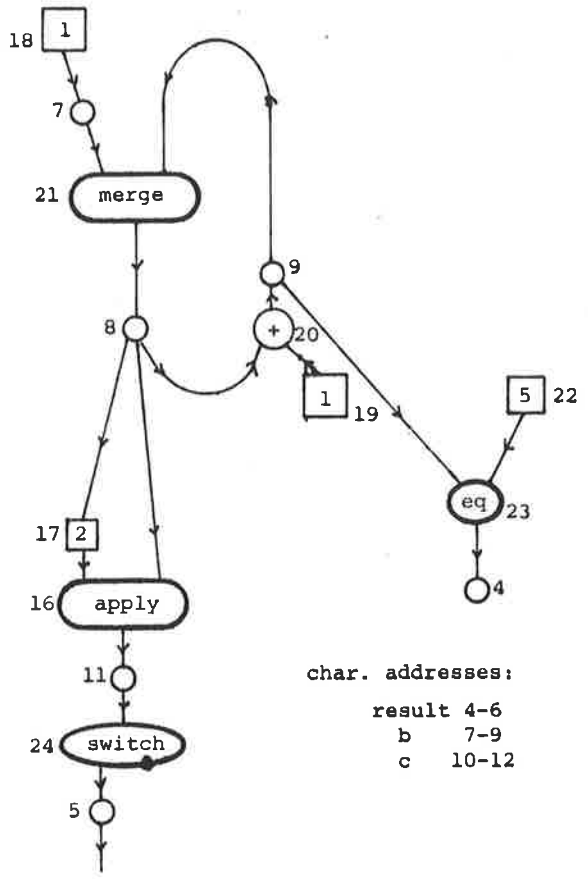


(a) Main program.



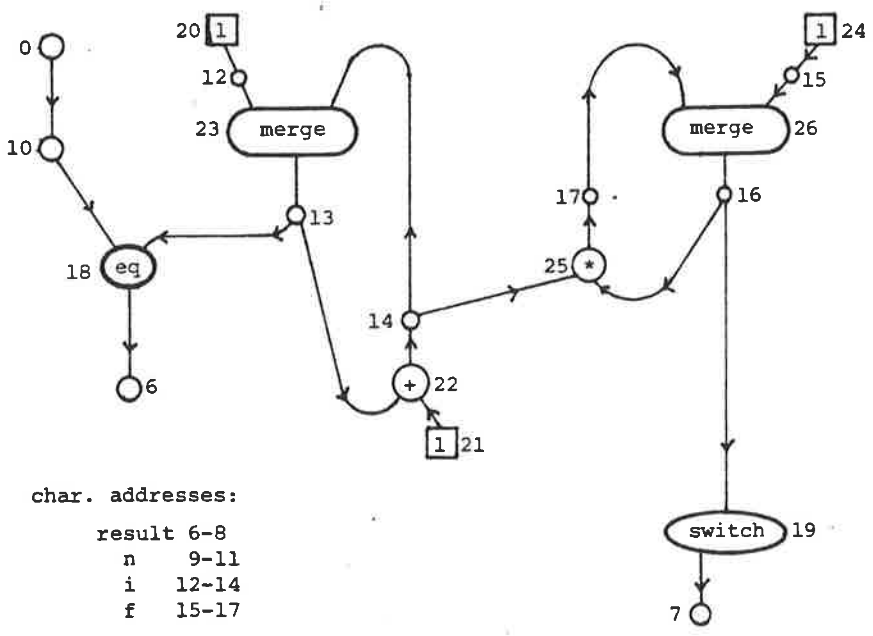
(b) **Define** clause.

Figure A3.2. Dependency graphs for program of Figure A3.1.



```
char. addresses:
      result 4-6
        b     7-9
        c    10-12
```

(a) Main program.



```
char. addresses:
      result 6-8
        n     9-11
        i    12-14
        f    15-17
```

(b) **Define** clause.

Figure A3.3. Initial data flow graphs for program of Figure A3.1.

code template entry:

⟨ *identifier, code address, gatelist, occlist, nplus* ⟩

gatelist entry:

[*arc, gating flag, occlist*]

occlist entry:

{ *link* }

(a) Format of a code template entry.

⟨ *result*, 4, *nil*, *nil*, *f* ⟩ ⟨ *b*, 9, [*b* 2 1,*ng*,{20.1}], *nil*, *t* ⟩

(b) Template for main program.

⟨ *n*, 10, *nil*, {18.1}, *f* ⟩ ⟨ *result*, 6, *nil*, *nil*, *f* ⟩
⟨ *i*, 14, [*i* 2 1,*g*,{22.1}], *nil*, *f* ⟩
⟨ *f*, 17, [*f* 2 1,*g*,{25.2}] [*i* 2 2,*ng*,{25.1}], *nil*,*f* ⟩

(c) Template for **define** clause.

Figure A3.4. Code template for program of Figure A3.1.

(4) Completed data flow graphs

Figure A3.5 shows the completed data flow graphs, produced by the code generator. Consider Figure A3.5(a), the main program, in which two significant changes from the initial graphs are evident; firstly, extra arcs to transmit values of the loop termination condition to control gates have been generated, and secondly, the circulator for *b* has been modified to include an FGATE instruction before the MERGE gate. From Figure A3.4, it can be seen that the *code address* component of the template entry for *result* indicates the address which delivers values of the termination condition; addresses of control gates are determined during a traversal of the *gatelist*. The template entry for *b* shows that the flag *nplus* is set. Gating rule 1 of Section 4.2.3 requires that an FGATE instruction be generated between the **next** network of *b* and the MERGE instruction of the circulator; accordingly, Figure A3.5(a) shows that the arc which originally linked the **next** network directly to the MERGE instruction has been replaced. The "current" value of *b*, used as input to the **next** network, is not gated.

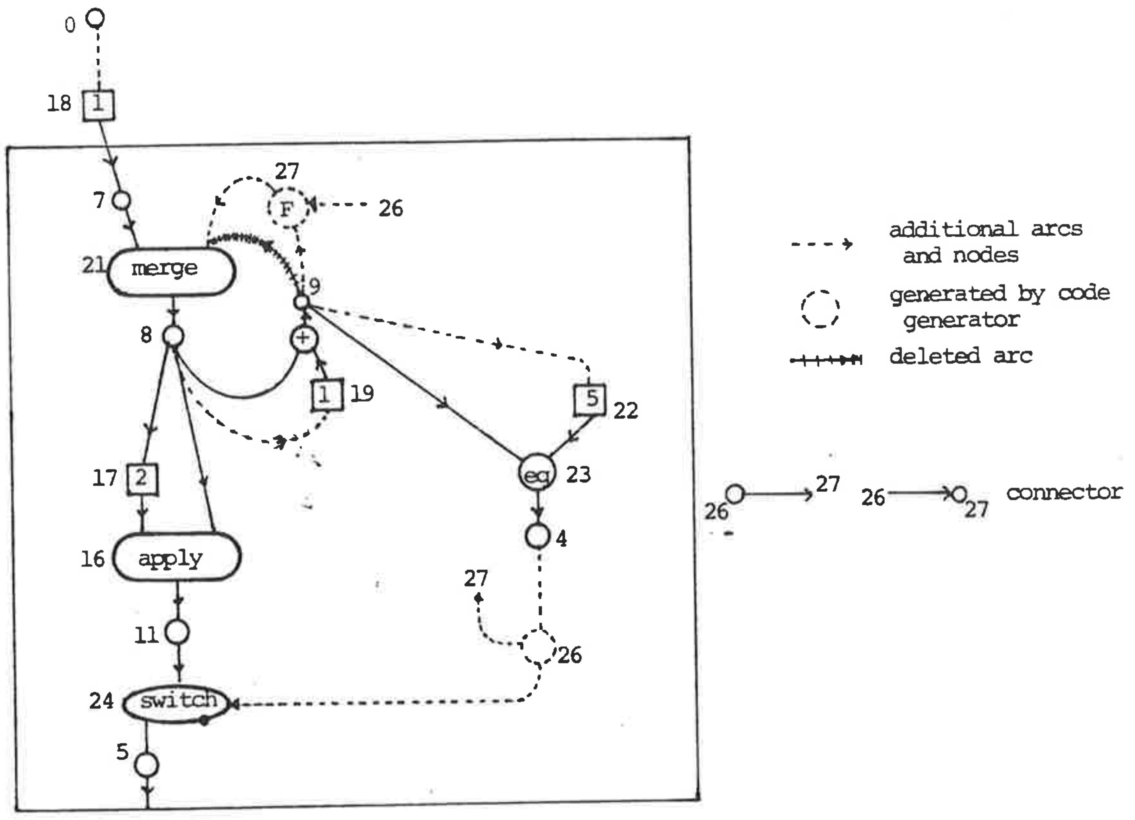
The graph for the **define** clause, Figure A3.5(b), shows that a simple circulator for *n* has been added. The value of the parameter *n* is frozen, hence constant, within the clause, and must be regenerated for each iteration of the loop. Control arcs and gating instructions have also been added; in this case, gating rule 3 determines the points at which FGATE instructions are inserted.

(5) Transformation of circulators to tail recursive form

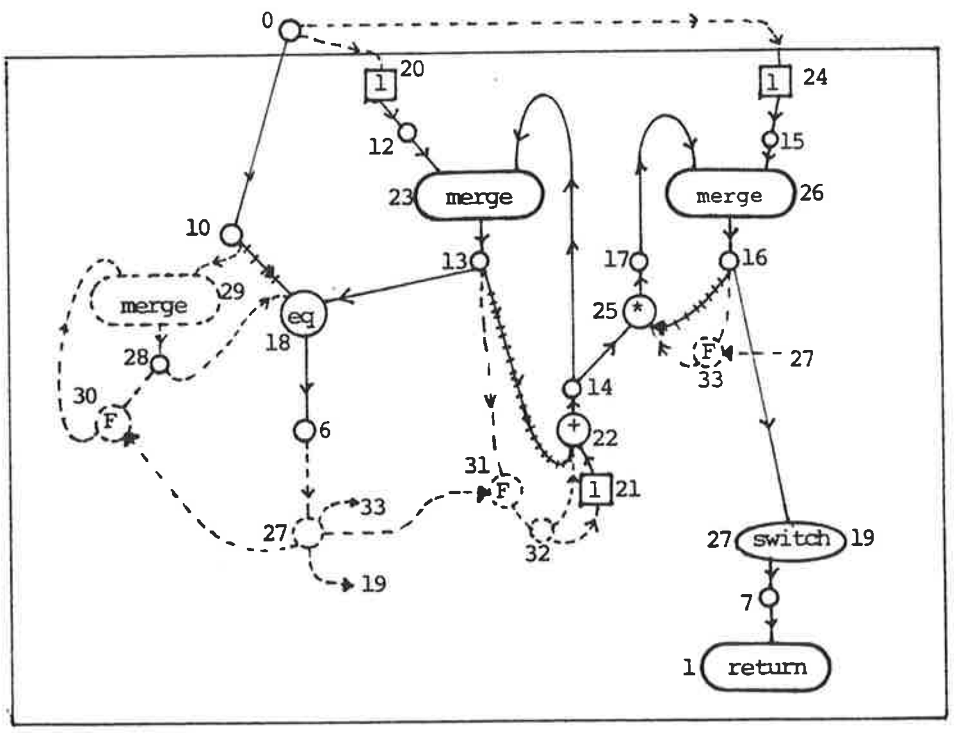
The final phase of the translation process is to eliminate all cycles by converting each loop in the graphs to tail recursive form, as described in Section 4.4.2. This process can be pictured in the following way:

(1) draw a box around a loop, in such a way that the only inputs to the box are the initial values of circulators, and the only output, the result returned from the loop
(2) form a new FT from the instructions inside the box
(3) in the original FT, replace the box with a suitable IAPPLY instruction

Each FT of Figure A3.6 uses one loop; hence, two additional function templates are created. The resulting four function templates of the target language program are shown in Figure A3.6, in a form suitable for execution by the data flow graph interpreter. The NULL instructions shown are never executed; they correspond to instructions which were left "outside the box" in step (1) above, and hence are not used in the FT for the loop. It is possible to eliminate these instructions by renumbering, but this step has been omitted to simplify comparison with previous templates.



(a) Main program.



(b) **Define** clause.

Figure A3.5. Final data flow graphs for program of Figure A3.1.

```
*0                                       Initiates loop of main program
0  Ident     0.0        :  2.2
1  Return    0.0        :  0.0
2  Con       =1  0.1    :  3.2
3  IApply    =1  2.1    :  4.1
4  Fin       3.1        :  0.0
5  End       0
*1                                       Loop FT for main program
0  Ident     0.0        :  3.1
1  Ident     0.0        :  7.1
2  Return    5.1        :  0.0
3  Incr      0.1 27.1   :  0.0
4  Ident     23.1       :  26.1
5  Ident     24.1       :  2.1
6  Ident                :
7  Ident     1.1        :  21.1
8  Ident     21.1       :  17.2 16.2 20.1 19.2
9  Ident     20.1       :  23.1 22.2 27.2
10 Ident                :
11 Ident     16.1       :  24.2
12 Ident                :
13 Ident                :
14 Ident                :
15 Ident                :
16 Apply     17.1 8.2   :  11.1
17 Con       =2 8.1     :  16.1
18 Null      0.0        :  0.0
19 Con       =1 8.4     :  20.2
20 Add       19.1 8.3   :  9.1
21 Ident     7.1        :  8.1
22 Con       =5 9.2     :  23.2
23 Eq        9.1 22.1   :  4.1
24 Switch    26.2 11.1  :  5.1 0.0
25 Null      0.0        :  0.0
26 Ident     4.1        :  27.1 24.1
27 FGate     26.1 9.3   :  3.2
28 End       0
*2                          Initiates loop of factorial function
0  Ident     0.0                 :  4.2 2.2 3.2
1  Return    4.1                 :  0.0
2  Con       =1 0.2              :  4.3
3  Con       =1 0.3              :  4.4
4  IApply    =3 0.1 2.1 3.1      :  1.1
5  End       0
*3                          Loop FT for factorial function
0  Ident     0.0                      :  5.1
1  Ident     0.0                      :  10.1
2  Ident     0.0                      :  12.1
3  Ident     0.0                      :  15.1
4  Return    7.1                      :  0.0
5  Incr      0.1 30.1 14.1 17.1       :  0.0
6  Ident     18.1                     :  27.1
7  Ident     19.1                     :  4.1
8  Ident                              :
9  Ident                              :
10 Ident     1.1                      :  28.1
11 Ident                              :
12 Ident     2.1                      :  23.1
13 Ident     23.1                     :  18.2 31.2
14 Ident     18.1                     :  5.3 25.1
15 Ident     3.1                      :  26.1
16 Ident     26.1                     :  33.2 19.2
17 Ident     25.1                     :  5.4
18 Eq        29.2 13.1                :  6.1
```

```
19 Switch     27.2 16.2           :  7.1
20 Null       0.0                 :  0.0
21 Con        =1 32.2             :  22.2
22 Add        32.1 21.1           :  14.1
23 Ident      12.1                :  13.1
24 Null       0.0                 :  0.0
25 Times      14.2 33.1           :  17.1
26 Ident      15.1                :  16.1
27 Ident      6.1                 :  30.1 19.1 31.1 33.1
28 Ident      10.1                :  29.1
29 Ident      28.1                :  30.2 18.1
30 FGate      27.1 29.1           :  5.2
31 FGate      27.3 13.2           :  32.1
32 Ident      31.1                :  22.1 21.2
33 FGate      27.4 16.1           :  25.2
34 End        0
```

Figure A3.6. Target language version of program of Figure A3.1.

Figure A3.7 shows a partial execution trace for the program of Figure A3.6. Several interesting points in the trace are indicated. The main program, FT 0, is initiated as the activation with Uid 4. At point 1, the main program loop is initiated, as activation 5; it will be noticed that the activities generated by the IAPPLY activity shown have an iteration number field, initially zero. Point 2 shows resumption of tracing at the initiation of the final iteration of the main program loop, FT 1; it can be seen that 4 is the value of *b* admitted to this final iteration, via the INCR activity [11,2,3]. At point 3, the **define** clause, FT 2, is invoked as activation 16. At point 4, the IAPPLY instruction initiates activation 17, the first iteration of the loop which computes *result* in the **define** clause. The first input, 3, indicates the FT which implements the loop; the remaining three inputs initialize the circulators of the loop, in this case with values 4, 1 and 1. The initiation of a subsequent iteration is shown at point 5. The final iteration of the **define** clause loop is initiated at point 6; the value admitted to the circulator for *f* is 24, or 4!. At point 7, this value has been returned, via the RETURN instruction of the final iteration, to the FT for the **define** clause, and thence to activation 15, the final iteration of the main program loop. The end of the trace, point 8, shows that the value has been returned to the main program, activation 4, and displayed using a FIN instruction.

```
Data Flow Graph Interpreter
Do you want tracing? y
Heap node 0 allocated as Fn
Heap node 1 allocated as Fn
Heap node 2 allocated as Fn
Heap node 3 allocated as Fn
Which Fn to run ? 0
Activating Fn with Uid: 0
Heap node 4 allocated as Ac
 ACTIVITY LIST
 [4,0]
        :

INTERPRETING ACTIVITY [4,3]
EXECUTING INSTRUCTION                                   (1)
 IApply      In :2 InC:1   Con: .1=1
                           Out:1 4.1
            Inputs 1 1
Activating Fn with Uid: 1
Heap node 5 allocated as Ac
Opnd 1 of Inst. 0 in Uid 5 = 1
Opnd 1 of Inst. 1 in Uid 5 = 1
Opnd 2 of Inst. 2 in Uid 5 = [4,3].0
            Outputs NU
 ACTIVITY LIST
 [5,0,1]  [5,0,0]
        :

INTERPRETING ACTIVITY [11,2,3]
EXECUTING INSTRUCTION                                   (2)
 Incr        In :2 InC:2   Con:
                           Out:1 0.0
            Inputs 1 4
Activating Fn with Uid: 1
Heap node 15 allocated as Ac
Opnd 1 of Inst. 0 in Uid 15 = 1
Opnd 1 of Inst. 1 in Uid 15 = 4
Opnd 2 of Inst. 2 in Uid 15 = [4,3].0
            Outputs NU
 ACTIVITY LIST
 [15,3,1]  [15,3,0]  [13,0,18]  [14,1,26]  [14,1,23]  [14,1,28]
        :

INTERPRETING ACTIVITY [15,3,16]
EXECUTING INSTRUCTION                                   (3)
 Apply       In :2 InC:2   Con:
                           Out:1 11.1
            Inputs 2 4
Activating Fn with Uid: 2
Heap node 16 allocated as Ac
Opnd 1 of Inst. 0 in Uid 16 = 4
Opnd 2 of Inst. 1 in Uid 16 = [15,16].0
            Outputs NU
 ACTIVITY LIST
 [16,0]  [13,0,22]  [14,1,7]  [15,3,20]
        :

INTERPRETING ACTIVITY [16,4]
EXECUTING INSTRUCTION                                   (4)
 IApply      In :4 InC:3   Con: .1=3
                           Out:1 1.1
            Inputs 3 4 1 1
Activating Fn with Uid: 3
Heap node 17 allocated as Ac
Opnd 1 of Inst. 0 in Uid 17 = 3
Opnd 1 of Inst. 1 in Uid 17 = 4
Opnd 1 of Inst. 2 in Uid 17 = 1
Opnd 1 of Inst. 3 in Uid 17 = 1
Opnd 2 of Inst. 4 in Uid 17 = [16,4].0
            Outputs NU


 ACTIVITY LIST
 [17,0,3]  [17,0,2]  [17,0,1]  [17,0,0]  [13,0,17]  [8,1,11]  [15,3,23]
        :
```

```
INTERPRETING ACTIVITY [17,0,5]
EXECUTING INSTRUCTION                                   (5)
 Incr         In :4 InC:4    Con:
                             Out:1 0.0
              Inputs 3 4 2 2
Activating Fn with Uid: 3
Heap node 19 allocated as Ac
Opnd 1 of Inst. 0 in Uid 19 = 3
Opnd 1 of Inst. 1 in Uid 19 = 4
Opnd 1 of Inst. 2 in Uid 19 = 2
Opnd 1 of Inst. 3 in Uid 19 = 2
Opnd 2 of Inst. 4 in Uid 19 = [16,4].0
              Outputs NU
 ACTIVITY LIST
 [19,1,3]  [19,1,2]  [19,1,1]  [19,1,0]  [18,1,17]
            :
INTERPRETING ACTIVITY [21,2,5]
EXECUTING INSTRUCTION                                   (6)
 Incr         In :4 InC:4    Con:
                             Out:1 0.0
              Inputs 3 4 4 24
Activating Fn with Uid: 3
Heap node 22 allocated as Ac
Opnd 1 of Inst. 0 in Uid 22 = 3
Opnd 1 of Inst. 1 in Uid 22 = 4
Opnd 1 of Inst. 2 in Uid 22 = 4
Opnd 1 of Inst. 3 in Uid 22 = 24
Opnd 2 of Inst. 4 in Uid 22 = [16,4].0
              Outputs NU
 ACTIVITY LIST
 [22,3,3]  [22,3,2]  [22,3,1]  [22,3,0]
            :
INTERPRETING ACTIVITY [22,3,4]
EXECUTING INSTRUCTION
 Return       In :1 InC:1    Con:
                             Out:1 0.0
              Inputs 24
Return via [16,4]
              Outputs 24
 ACTIVITY LIST
 [16,1]
INTERPRETING ACTIVITY [16,1]
EXECUTING INSTRUCTION                                   (7)
 Return       In :1 InC:1    Con:
                             Out:1 0.0
              Inputs 24
Return via [15,3,16]
              Outputs 24
 ACTIVITY LIST
 [15,3,11]
            :
INTERPRETING ACTIVITY [15,3,2]
EXECUTING INSTRUCTION
 Return       In :1 InC:1    Con:
                             Out:1 0.0
              Inputs 24
Return via [4,3]
              Outputs 24
 ACTIVITY LIST
 [4,4]


INTERPRETING ACTIVITY [4,4]
EXECUTING INSTRUCTION                                   (8)
 Fin          In :1 InC:1    Con:
                             Out:1 0.0
              Inputs 24
Result at output 1 = 24
              Outputs NU
```

Figure A3.7. Execution trace of program of Figure A3.6.

APPENDIX 4

AN EXAMPLE OF TRANSLATION OF AN LX PROGRAM

In this Appendix, the production of data flow graphs fom a simple LX program is described. The program, shown in Figure A4.1, is a factorial program which defines a history of factorials $[\![f]\!]$ in terms of its own future; a similar program was used in §2.3.2. The following histories satisfy the definitions of the program:

$$\begin{aligned}
[\![m]\!] &= \langle\, 3,\, 3,\, 3,\, 3,\, \ldots \,\rangle \\
[\![n]\!] &= \langle\, 3,\, 2,\, 1,\, 0,\, \ldots \,\rangle \\
[\![f]\!] &= \langle\, 6,\, 2,\, 1,\, 1,\, \ldots \,\rangle \\
[\![result]\!] &= \langle\, 6,\, 2,\, 1,\, 1,\, \ldots \,\rangle.
\end{aligned}$$

```
prog fact  global  int m;
   int n,  f,  result;
   n =  m fby n−1;
   f = if   n <  2
         then 1
         else n * next f
         eif;
   result =  f;
eprog
```

Figure A4.1. A factorial program.

The FTs produced from the program are shown in Figure A4.2. Heap node 0 is a standard driver which operates in a continuous cycle, of accepting a number ***d*** from the terminal, invoking the main program with ***d*** as a demand number, and displaying $[\![result]\!]_d$ at the terminal. FTs 1, 2 and 3 respectively are loaded with every program, and are used in the implementation of the operators **asa_then_easa**, **wvr_then_ewvr** and **upon_then_eupon**. A use of one of these operators is compiled as an invocation of the appropriate FT; the FT for **asa_then_easa** is included for information, although the construct is not used in this particular program.

The FT for the LX program, constructed as described in §5.3.3.1, is shown at heap node 4. The principal purpose of this FT is to use the supplied demand number to construct a second demand, as shown at instruction 7, which is then used to invoke the FT for *result*, at instruction 8. Instructions 3 to 6 show the construction of the component *instance* of this second demand. As indicated by the constant input operand of instruction 6, the component *text* of the instance is FT 4, the LX main program. The main program itself is the only clause used, hence only one instance is created. The scheme given in §5.3.3.1 requires, for an "external instance", an ET which includes a list of global variables of the main program, intended for use in resolving uses of such variables. However, for this particular LX program, the compiler can resolve such uses directly, and the ET is not needed. Hence, it is created with an empty list, at heap node 5, and a dummy instance.

FT 6 is invoked to produce a value of $[\![m]\!]$. The external interface required to obtain such values is implemented in the special instruction *Prompt*, which displays its inputs, a character string and the demand number, on the screen, and accepts a value in response. Such values are buffered to prevent multiple requests for the value at a given history index.

Both FT 7 and FT 8, representing respectively the definitions of n and f, are constructed as described in §5.3.3.3. The *DNumI* instruction (number 11) of FT 8 performs the demand number incrementation required to obtain values of **next** f. The SWITCH instructions 14 and 15 simulate a TGATE and an FGATE instruction. As mentioned in §3.2, the MERGE instruction has two input operands, but fires on arrival of an input at either operand. Hence, its *incount*, the number of values yet to arrive to enable the instruction, is 1; to ensure that the interpreter initializes this count correctly, only one input source is shown.

```
*0 DRIVER
0 Ident      0.0      :  2.2
1 Return     1.1      :
2 Start      =0       :  3.2
3 Apply      =4  2.1  :  4.1
4 Ident      3.1      :  6.1 5.2
5 Apply      =0  4.2  :  1.1
6 Fin        4.2      :
7 End        0
*1  ASA
0  Ident      0.0       :  4.1
1  Ident      0.0       :  5.1
2  Ident      0.0       :  8.1
3  Return     10.1      :
4  Incr       0.1 7.2 11.1 :
5  Merge      1.1       :  6.1
6  Ident      5.1       :  7.2 12.1
7  Switch     13.1 6.1  :  0.0 4.2
8  Merge      2.1       :  9.1
9  Ident      8.1       :  12.2 10.2
10 Switch     13.2 9.2  :  3.1 11.1
11 DNumI      10.2      :  4.3
12 Apply      6.2 9.1   :  13.1
13 Ident      12.1      :  7.1 10.1
14 End        0
*2 WVR
*3 UPON
*4 program
0 Ident      0.0 :  2.1
1 Return     8.1 :
2 Ident      0.1 :  3.2 5.1 7.1
3 Con        =5 2.1 :  4.1
4 LUpd       3.1 =I[0] :  5.2
5 ETCons     2.2 4.1 :  6.2
6 ICons      =FT[4] 5.1 :  7.2
7 DCons      2.3 6.1 :  8.2
8 Apply      9 7.1 :  1.1
9 End        0
*5                                        empty list
*6                                        FT for "m"
0 Ident      0.0 :  2.1
1 Return     3.1 :
2 DNum       0.1 :  3.2
3 Prompt     =M   2.1 :  1.1
4 End        0
*7                                        FT for "n"
0 Ident      0.0 :  7.1
1 Return     14.1 :
2 Apply      =6 11.1 :  14.1
3 Ident      13.1 :  4.2 6.2
4 Apply      =7 3.1 :  5.1
5 Sub        4.1 6.1 :  14.1
6 Con        =1 3.2 :  5.2
7 Ident      0.1 :  8.1 11.2 12.2
8 DNum       7.1 :  9.2
9 Eq         =0 8.1 :  10.1
10 Ident     9.1 :  11.1 12.1
11 Switch    10.1 7.2 :  2.2
12 Switch    10.2 7.3 :  0.0 13.1
13 DNumD     12.1 :  3.1
14 Merge     2.1 :  1.1
15 End       0
```

```
*8                                              Defn.  of f
0 Ident        0.0 :  12.1
1 Return       16.1 :
2 Ident        12.1 :  3.2 5.2
3 Apply        =7 2.1 :  4.1
4 Lt           3.1 5.1 :  13.1
5 Con          =2 2.2 :  4.2
6 Con          =1 14.1 :  16.1
7 Ident        15.2 :  8.2 11.1
8 Apply        =7 7.1 :  9.1
9 Times        8.1 10.1 :  16.1
10 Apply       =8 11.1 :  9.2
11 DNumI       7.2 :  10.2
12 Ident       0.1 :  2.1 14.2 15.2
13 Ident       4.1 :  14.1 15.1
14 Switch      13.1 12.2 :  6.2 0.0
15 Switch      13.2 12.3 :  0.0 7.1
16 Merge       6.1 :  1.1
17 End         0
*9                                              FT for "result"
0 Ident       0.0 :  2.2
1 Return      2.1 :
2 Apply       =8 0.1 :  1.1
3 End         0
```

Figure A4.2. Function templates produced from LX program of Figure A4.1.

Figure A4.3 shows the propagation of demands to various definitions and expressions in the program. The arcs representing demand propagation are labelled with the demand number propagated. The return of values is also shown, by labelling arcs with the returned value.

An extract from a trace of the execution of the program of Figure A4.2 is presented in Figure A4.4. An explanation of each of the points labelled on the trace follows:

1: The inputs of the *DCons* instruction are a demand number 0 and an instance at heap node 14, which comprises an FT at node 4 and an ET at node 13; the output shows each component of the demand.

2: Heap node 16 is created as an activation of f, supplied with demand number 0.

3: Heap node 17 is an activation of n, with demand number 0.

4: Shows extraction of the demand number propagated to activation 17, and its comparison with 0, as required to implement **fby**.

5: The FT for m has been invoked, and the value 3 accepted from the terminal and returned from the activation.

6: Shows execution of the test $n < 2$.

7: The demand number has been incremented to 1, and f invoked, as required to implement **fby**.

8: Considerably later in execution, f is invoked with demand number 2, creating activation 26.

9: In activation 26, the test $n < 2$ succeeds, hence the value 1 is returned and multiplied by $n_1 = 2$.

10: Shows a later point on the VRP.

11: The value $3! = 6$ is returned to the main program, via activations of f and *result*.

12: The value 6 is displayed on the terminal.

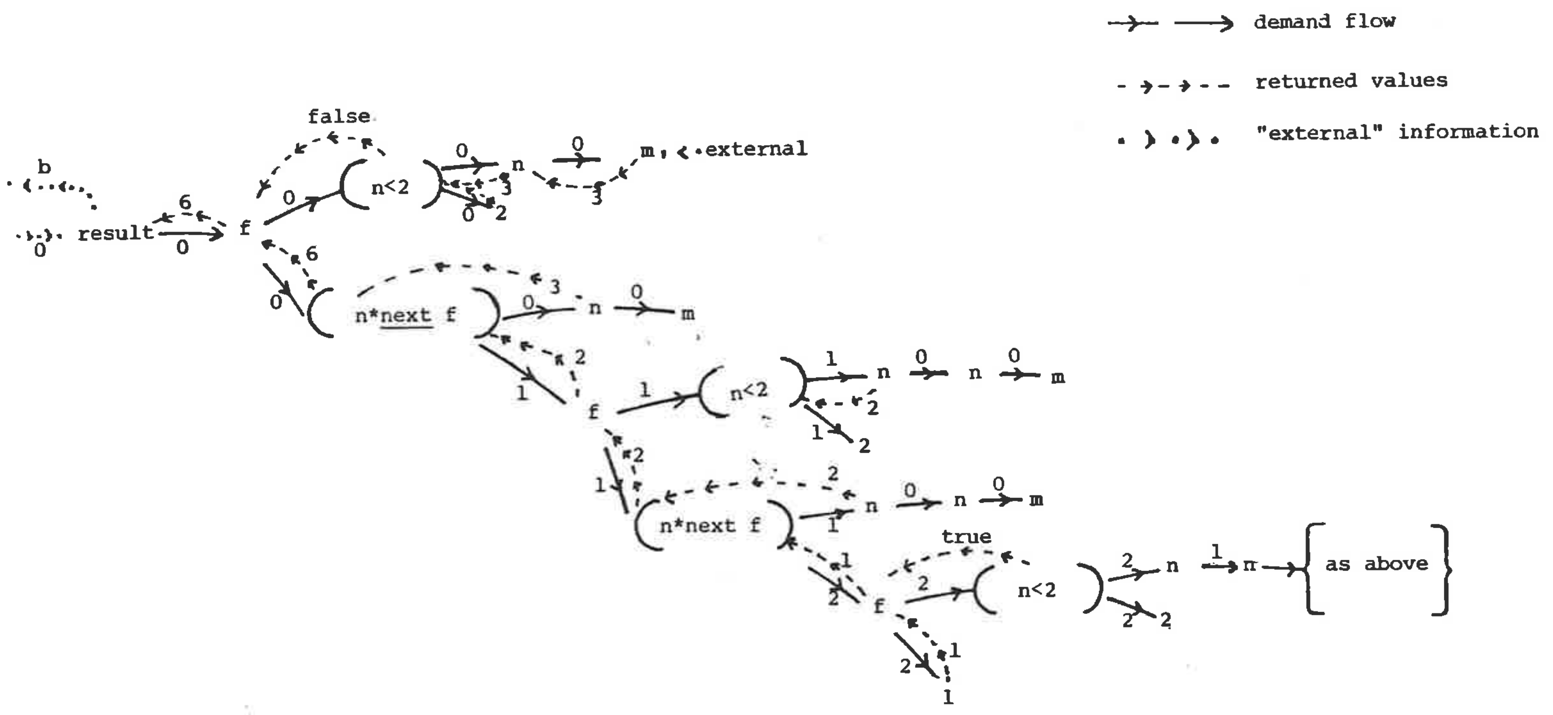


Figure A4.3. Demand flow in the example of Figure A4.1.

```
Data Flow Graph Interpreter
Do you want tracing? y
Heap node 0 allocated as Fn
Heap node 1 allocated as Fn
Heap node 2 allocated as Fn
Heap node 3 allocated as Fn
Heap node 4 allocated as Fn
Heap node 5 allocated as L1
Heap node 6 allocated as Fn
Heap node 7 allocated as Fn
Heap node 8 allocated as Fn
Heap node 9 allocated as Fn
Heap read 10 nodes
Which Fn to run ? 0
Activating Fn with Uid: 0
Heap node 10 allocated as Ac
 ACTIVITY LIST
 [10,0]
INTERPRETING ACTIVITY [10,0]
EXECUTING INSTRUCTION
 Ident       In :1 InC:1   Con:
                           Out:1 2.1
            Inputs 0
            Outputs 0
 ACTIVITY LIST
 [10,2]
INTERPRETING ACTIVITY [10,2]
EXECUTING INSTRUCTION
 Start       In :1 InC:1   Con:
                           Out:1 3.2
            Inputs 0
New value for input 1: 0
            Outputs 0
 ACTIVITY LIST
 [10,3]
INTERPRETING ACTIVITY [10,3]
EXECUTING INSTRUCTION
 Apply       In :2 InC:1   Con: .1=4
                           Out:1 4.1
            Inputs 4 0
Activating Fn with Uid: 4
Heap node 11 allocated as Ac
Opnd 1 of Inst. 0 in Uid 11 = 0
Opnd 2 of Inst. 1 in Uid 11 = [10,3].0
            Outputs NU
 ACTIVITY LIST
 [11,0]
       :
INTERPRETING ACTIVITY [11,4]
EXECUTING INSTRUCTION
 LUpd        In :2 InC:1   Con: .2=I[0]
                           Out:1 5.2
            Inputs 5 I[0]
Heap node 12 allocated as L1
List 0 elts.
            Outputs L[12]
 ACTIVITY LIST
 [11,5]
INTERPRETING ACTIVITY [11,5]
EXECUTING INSTRUCTION
 ETCons      In :2 InC:2   Con:
                           Out:1 6.2
            Inputs 0 L[12]
Heap node 13 allocated as ET
            Outputs ET[13]
 ACTIVITY LIST
 [11,6]
INTERPRETING ACTIVITY [11,6]
EXECUTING INSTRUCTION
 ICons       In :2 InC:1   Con: .1=FT[4]
                           Out:1 7.2
            Inputs FT[4] ET[13]
Heap node 14 allocated as Ic
            Outputs I[14]=[4,13]
 ACTIVITY LIST
 [11,7]
INTERPRETING ACTIVITY [11,7]                        (1)
EXECUTING INSTRUCTION
 DCons      In :2 InC:2   Con:
                          Out:1 8.2
            Inputs 0 I[14]=[4,13]
            Outputs D[0,14]
 ACTIVITY LIST
 [11,8]
INTERPRETING ACTIVITY [11,8]
EXECUTING INSTRUCTION
 Apply      In :2 InC:1   Con:  .1=9
                          Out:1 1.1
            Inputs 9 D[0,14]
Activating Fn with Uid: 9
Heap node 15 allocated as Ac
Opnd 1 of Inst. 0 in Uid 15 = D[0,14]
Opnd 2 of Inst. 1 in Uid 15 = [11,8].0
            Outputs NU
 ACTIVITY LIST
 [15,0]
INTERPRETING ACTIVITY [15,0]
EXECUTING INSTRUCTION
 Ident      In :1 InC:1   Con:
                          Out:1 2.2
            Inputs D[0,14]
            Outputs D[0,14]
 ACTIVITY LIST
 [15,2]
INTERPRETING ACTIVITY [15,2]                        (2)
EXECUTING INSTRUCTION
 Apply      In :2 InC:1   Con:  .1=8
                          Out:1 1.1
            Inputs 8 D[0,14]
Activating Fn with Uid: 8
Heap node 16 allocated as Ac
Opnd 1 of Inst. 0 in Uid 16 = D[0,14]
Opnd 2 of Inst. 1 in Uid 16 = [15,2].0
            Outputs NU
 ACTIVITY LIST
 [16,0]
      :
INTERPRETING ACTIVITY [16,3]                        (3)
EXECUTING INSTRUCTION
 Apply      In :2 InC:1   Con:  .1=7
                          Out:1 4.1
            Inputs 7 D[0,14]
Activating Fn with Uid: 7
Heap node 17 allocated as Ac
Opnd 1 of Inst. 0 in Uid 17 = D[0,14]
Opnd 2 of Inst. 1 in Uid 17 = [16,3].0
            Outputs NU
 ACTIVITY LIST
 [17,0]  [16,5]
      :
INTERPRETING ACTIVITY [17,8]
EXECUTING INSTRUCTION
 DNum       In :1 InC:1   Con:
                          Out:1 9.2
            Inputs D[0,14]
            Outputs 0
 ACTIVITY LIST
 [17,9]
INTERPRETING ACTIVITY [17,9]                        (4)
EXECUTING INSTRUCTION
 Eq         In :2 InC:1   Con:  .1=0
                          Out:1 10.1
            Inputs 0 0
            Outputs T
 ACTIVITY LIST
 [17,10]
       :
       :
INTERPRETING ACTIVITY [17,2]
EXECUTING INSTRUCTION
 Apply       In :2 InC:1   Con: .1=6
                           Out:1 14.1
            Inputs 6 D[0,14]
Activating Fn with Uid: 6
Heap node 18 allocated as Ac
Opnd 1 of Inst. 0 in Uid 18 = D[0,14]
Opnd 2 of Inst. 1 in Uid 18 = [17,2].0
            Outputs NU
 ACTIVITY LIST
 [18,0]
       :
       :
INTERPRETING ACTIVITY [18,3]
EXECUTING INSTRUCTION
 Prompt      In :2 InC:1   Con: .1=M
                           Out:1 1.1
            Inputs M    0
M   at 0: 3
            Outputs 3
 ACTIVITY LIST
 [18,1]
INTERPRETING ACTIVITY [18,1]                          (5)
EXECUTING INSTRUCTION
 Return      In :1 InC:1   Con:
                           Out:0
            Inputs 3
Return via [17,2]
            Outputs
 ACTIVITY LIST
 [17,14]
INTERPRETING ACTIVITY [17,14]
EXECUTING INSTRUCTION
 Merge       In :1 InC:1   Con:
                           Out:1 1.1
            Inputs 3
            Outputs 3
 ACTIVITY LIST
 [17,1]
INTERPRETING ACTIVITY [17,1]
EXECUTING INSTRUCTION
 Return      In :1 InC:1   Con:
                           Out:0
            Inputs 3
Return via [16,3]
            Outputs
 ACTIVITY LIST
 [16,4]
INTERPRETING ACTIVITY [16,4]                          (6)
EXECUTING INSTRUCTION
 Lt          In :2 InC:2   Con:
                           Out:1 13.1
            Inputs 3 2
            Outputs F
 ACTIVITY LIST
 [16,13]
       :
       :
INTERPRETING ACTIVITY [16,11]
EXECUTING INSTRUCTION
 DNumI       In :1 InC:1   Con:
                           Out:1 10.2
            Inputs D[0,14]
            Outputs D[1,14]
 ACTIVITY LIST
 [16,10]
```

```
INTERPRETING ACTIVITY [16,10]                          (7)
EXECUTING INSTRUCTION
 Apply       In :2 InC:1    Con: .1=8
                            Out:1 9.2
            Inputs 8 D[1,14]
Activating Fn with Uid: 8
Heap node 20 allocated as Ac
Opnd 1 of Inst. 0 in Uid 20 = D[1,14]
Opnd 2 of Inst. 1 in Uid 20 = [16,10].0
            Outputs NU
 ACTIVITY LIST
 [20,0]  [19,7]
      ⋮

INTERPRETING ACTIVITY [20,10]                          (8)
EXECUTING INSTRUCTION
 Apply       In :2 InC:1    Con: .1=8
                            Out:1 9.2
            Inputs 8 D[2,14]
Activating Fn with Uid: 8
Heap node 26 allocated as Ac
Opnd 1 of Inst. 0 in Uid 26 = D[2,14]
Opnd 2 of Inst. 1 in Uid 26 = [20,10].0
            Outputs NU
 ACTIVITY LIST
 [26,0]  [25,7]
      ⋮

INTERPRETING ACTIVITY [26,4]
EXECUTING INSTRUCTION
 Lt          In :2 InC:2    Con:
                            Out:1 13.1
            Inputs 1 2
            Outputs T
 ACTIVITY LIST
 [26,13]
      ⋮

INTERPRETING ACTIVITY [26,1]
EXECUTING INSTRUCTION
 Return      In :1 InC:1    Con:
                            Out:0
            Inputs 1
Return via [20,10]
            Outputs
 ACTIVITY LIST
 [20,9]
INTERPRETING ACTIVITY [20,9]                           (9)
EXECUTING INSTRUCTION
 Times       In :2 InC:2    Con:
                            Out:1 16.1
            Inputs 2 1
            Outputs 2
 ACTIVITY LIST
 [20,16]
      ⋮

INTERPRETING ACTIVITY [20,1]
EXECUTING INSTRUCTION
 Return      In :1 InC:1    Con:
                            Out:0
            Inputs 2
Return via [16,10]
            Outputs
 ACTIVITY LIST
 [16,9]
INTERPRETING ACTIVITY [16,9]                           (10)
EXECUTING INSTRUCTION
 Times       In :2 InC:2    Con:
                            Out:1 16.1
            Inputs 3 2
            Outputs 6
 ACTIVITY LIST
 [16,16]
      :
      :
INTERPRETING ACTIVITY [16,1]
EXECUTING INSTRUCTION
 Return      In :1 InC:1    Con:
                            Out:0
             Inputs 6
Return via [15,2]
             Outputs
 ACTIVITY LIST
 [15,1]
INTERPRETING ACTIVITY [15,1]
EXECUTING INSTRUCTION
 Return      In :1 InC:1    Con:
                            Out:0
             Inputs 6
Return via [11,8]
             Outputs
 ACTIVITY LIST
 [11,1]
INTERPRETING ACTIVITY [11,1]                     (11)
EXECUTING INSTRUCTION
 Return      In :1 InC:1    Con:
                            Out:0
             Inputs 6
Return via [10,3]
             Outputs
 ACTIVITY LIST
 [10,4]
      :
      :
INTERPRETING ACTIVITY [10,6]                     (12)
EXECUTING INSTRUCTION
 Fin         In :1 InC:1    Con:
                            Out:0
             Inputs 6
Result at output 1 = 6
             Outputs
```

Figure A4.4. A trace of the execution of the program of Figure A4.2.

APPENDIX 5

THE TRANSLATION OF A PROGRAM USING THE HYBRID SCHEME

This Appendix completes the description of the translation of the program shown in Figure 6.7. The graphical form of the DFC is shown in Figure 6.8. Figure A5.1 shows the textual representation of a data flow program which implements the program given in Figure 6.7. In this case, the data flow program was constructed manually from graphs produced by the implementations described in Chapters 4 and 5.

Function templates 0 and 5 represent the DFC. FT 0 contains that part of the graph which is "outside the box" (a term explained in Appendix 3) shown in Figure 6.8, as well as a FIN instruction used to display the result of the program. Note that FT0 invokes the main program (FT 5) once, to provide the "external trigger" mentioned in §6.4. If the program required external input, for example if f was global, then FT 0 would use a START instruction to obtain one input value, invoke FT 5 with that value, and then recursively invoke itself (FT 0) to obtain another input value (see §3.3.2).

FT 5 contains the main body of the loop shown in Figure 6.8, after it has been transformed to tail recursive form. FT 6 is the representation in the DDC of the DF variable f. It can be seen that it invokes FT 4, which implements the *Access* network described in §6.3.2; a parameter of the invocation is the Uid 7, the heap node which contains the structure value H_f. All remaining FTs implement the DDC, and are as generated by the implementation of §5.3.

Explanation of each of the labelled points on Figure A5.1 follows.

1: Instructions 3 and 4 implement $FirstH_f$ of Figure 6.8.

2: Instructions 18 and 19 implement *DI* of Figure 6.8; inputs 3 and 4 of *DCons* implement actual parameter transmission, which is not considered in detail in this thesis.

3: Instructions 30 to 33 implement $NextH_f$ of Figure 6.8.

4: Instructions 7 and 8 are concerned with actual parameter transmission.

5: Heap nodes 13, 14 and 19 are lists of global variables and formal parameters created by the LX compiler.

6: Instruction 3 is an invocation of FT 2, *wvr*; the second operand 17 is an FT which computes a value of the condition of the **wvr** definition, and is invoked within FT 3 when required.

```
*0                                              Initiates loop of main program
0  Ident    0.0           :  2.2 3.2 5.2
1  Return   7.1           :  0.0
2  Con      =5 0.1        :  6.2 4.2
3  Con      =7 0.2        :  4.1 6.3                    (1)
4  MkL      3.1 2.2       :  0.0
5  Con      =0 0.3        :  6.4
6  IApply   =5 2.1 3.2 5.1 :  7.1
7  Fin      6.1           :  1.1
9  End      0
*1  ASA
0  Ident    0.0      :  4.1
1  Ident    0.0      :  5.1
2  Ident    0.0      :  8.1
3  Return   10.1     :
4  Incr     0.1 7.2 11.1 :
5  Merge    1.1      :  6.1
6  Ident    5.1      :  7.2 12.1
7  Switch   13.1 6.1 :  0.0 4.2
8  Merge    2.1      :  9.1
9  Ident    8.1      :  12.2 10.2
10 Switch   13.2 9.2 :  3.1 11.1
11 DNumI    10.2     :  4.3
12 Apply    6.2 9.1  :  13.1
13 Ident    12.1     :  7.1 10.1
14 End      0
*2 WVR
0  Ident    0.0      :  6.1
1  Ident    0.0      :  7.1
2  Ident    0.0      :  10.1
3  Ident    0.0      :  14.1
4  Ident    0.0      :  18.1
5  Return   12.1     :
6  Incr     0.1 9.2 13.1 17.1 20.2 :
7  Merge    1.1      :  8.1
8  Ident    7.1      :  9.2 11.2
9  Switch   22.4 8.1 :  0.0 6.2
10 Merge    2.1      :  11.3
11 IApply   =1 8.2 10.1 :  12.2
12 Switch   22.3 11.1 :  5.1 13.1
13 DNumI    12.2     :  6.3
14 Merge    3.1      :  15.1
15 Ident    14.1     :  16.2 21.1
16 Switch   22.1 15.1 :  0.0 17.2
17 Add      =1 16.2 :  6.4
18 Merge    4.1      :  19.1
19 Ident    18.1     :  23.1 20.2
20 Switch   22.2 19.2 :  0.0 6.5
21 Eq       15.2 23.1 :  22.1
22 Ident    21.1     :  9.1 12.1 16.1 20.1
23 DNum     19.1     :  21.2
24 End      0
*3 UPON
*4 ACCESS
```

```
0  Ident    0.0             :  4.1
1  Ident    0.0             :  5.1
2  Ident    0.0             :  9.1
3  Return   16.1            :  0.0
4  Incr     0.1 8.1 12.1    :  0.0
5  Ident    1.1             :  6.1
6  Ident    5.1             :  7.2 15.2
7  FGate    14.2 6.1        :  8.1
8  R        7.1             :  4.2
9  Ident    9.1             :  10.1
10 Ident    9.1             :  11.2 13.1
11 FGate    14.3 10.1       :  12.1
12 Sub      11.1 =1         :  4.3
13 Eq       10.2 =0         :  14.1
14 Ident    13.1            :  15.1 7.1 11.1
15 Switch   14.1 6.2        :  16.1 0.0
16 L        15.1            :  3.1
17 End      0
*5                                            DFC FT for main program
0  Ident    0.0                   :  5.1
1  Ident    0.0                   :  9.1
2  Ident    0.0                   :  28.1
3  Ident    0.0                   :  12.1
4  Return   7.1                   :  0.0
5  Incr     0.1 11.2 17.1 14.1    :  0.0
6  Ident    23.1                  :  24.1 27.1 35.1 38.1
7  Ident    24.1                  :  4.1
8  Ident                          :
9  Ident    1.1                   :  26.1
10 Ident    26.1                  :  27.2
11 Ident    25.1                  :  33.2 5.2
12 Ident    3.1                   :  37.1
13 Ident    37.1                  :  18.1 20.1 38.2
14 Ident    36.1                  :  5.4
15 Ident    28.1                  :  34.1
16 Ident    34.1                  :  35.2
17 Ident    31.3                  :  5.3
18 ETCons   13.1 =0               :  19.2                (2)
19 ICons    =5 18.1               :  20.2
20 DCons    13.2 19.1 =0 =0       :  21.2
21 Apply    =9 20.1               :  22.1
22 Ident    21.1                  :  23.1 24.2
23 Gt       22.1 =500             :  6.1
24 Switch   6.1 22.2              :  7.1 0.0
25 Times    27.1 =5               :  11.1
26 Ident    9.1                   :  10.1
27 FGate    6.2 10.1              :  25.1
28 Ident    2.1                   :  15.1
29 Null                           :
30 Ident    2.1                   :  31.1 32.1           (3)
31 Pair     30.1                  :  32.2 33.1 17.1
32 MkR      30.1 31.1             :  0.0
33 MkL      31.2 11.1             :  0.0
34 Ident    15.1                  :  16.1
35 FGate    6.3 16.1              :  30.1
36 Add      38.1 =1               :  14.1
37 Ident    12.1                  :  13.1
38 FGate    6.4 13.3              :  36.1
39 End      0
*6                                                    DDC FT for "f"
0  Ident    0.0                   :  2.1
1  Return   3.1                   :  0.0
2  DNum     0.1                   :  3.3
3  IApply   =4 =7 2.1             :  1.1
```

```
4  End        0
*7                                              Structure Hf
*8                                              DDC FT for "b"
0  Ident      0.0                 :  7.1
1  Return     14.1                :  0.0
2  Con        =1 11.1             :  14.1
3  Ident      13.1                :  4.2 6.2
4  Apply      =8 3.1              :  5.1
5  Add        4.1 6.1             :  14.1
6  Con        =1 3.2              :  5.2
7  Ident      0.1                 :  8.1 11.2 12.2
8  DNum       7.1                 :  9.2
9  Eq         =0 8.1              :  10.1
10 Ident      9.1                 :  11.1 12.1
11 Switch     10.1 7.2            :  2.2 0.0
12 Switch     10.2 7.3            :  0.0 13.1
13 DNumD      12.2                :  3.1
14 Merge      2.1                 :  1.1
15 End        0
*9                                              DDC FT for "c"
0  Ident      0.0                 :  2.1
1  Return     6.1                 :  0.0
2  Ident      0.1                 :  3.1 4.1
3  DNum       2.1                 :  5.1
4  DInst      2.2                 :  5.2 5.3
5  DCons      3.1 4.1 4.2 =19     :  6.2
6  Apply      =10 5.1             :  1.1
7  End        0
*10                                             DDC FT for "filter"
0  Ident      0.0                 :  2.1
1  Return     14.1                :  0.0
2  Ident      0.1                 :  3.2 4.1 5.1 7.1 8.1
3  Con        =14 2.1             :  6.1
4  DNum       2.2                 :  11.1 13.1
5  DInst      2.3                 :  6.2
6  LUpd       3.1 5.1             :  10.2
7  DAPLi      2.4                 :  9.1 13.4          (4)
8  DAPi       2.5                 :  9.2 13.3
9  LUpd       7.1 8.1             :  10.1
10 LAppend    9.1 6.1             :  11.2
11 ETCons     4.1 10.1            :  12.2
12 ICons      =10 11.1            :  13.2
13 DCons      4.2 12.1 8.2 7.2    :  14.2
14 Apply      =15 13.1            :  1.1
15 End        0
*11                                             DDC FT for "p"
0  Ident      0.0                 :  2.1
1  Return     13.1                :  0.0
2  Ident      0.1                 :  3.1 10.1 11.1 12.1
3  DInst      2.1                 :  4.1
4  IET        3.1                 :  5.1
5  ETList     4.1                 :  6.1
6  LFind      5.1 =11             :  7.1 8.1
7  LUser      6.1                 :  9.2
8  LNew       6.2                 :  13.1
9  DCons      10.1 7.1 11.1 12.1  :  13.2
10 DNum       2.2                 :  9.1
11 DAPi       2.3                 :  9.3
12 DAPLi      2.4                 :  9.4
13 Apply      8.1 9.1             :  1.1
14 End        0
*12                                             DDC FT for global "f"
0  Ident      0.0                 :  2.1
```

```
1  Return   13.1                 :  0.0
2  Ident    0.1                  :  3.1 10.1 11.1 12.1
3  DInst    2.1                  :  4.1
4  IET      3.1                  :  5.1
5  ETList   4.1                  :  6.1
6  LFind    5.1 =12              :  7.1 8.1
7  LUser    6.1                  :  9.2
8  LNew     6.2                  :  13.1
9  DCons    10.1 7.1 11.1 12.1   :  13.2
10 DNum     2.2                  :  9.1
11 DAPi     2.3                  :  9.3
12 DAPLi    2.4                  :  9.4
13 Apply    8.1 9.1              :  1.1
14 End      0
*13                                                    List       (5)
*14                                                    List
*15                                                    DDC FT for 'result'
0  Ident    0.0                  :  2.1
1  Return   4.1                  :  0.0
2  Ident    0.1                  :  3.2 5.2
3  Apply    =16 2.1              :  4.1
4  Add      3.1 5.1              :  1.1
5  Apply    =12 2.2              :  4.2
6  End      0
*16                                                    DDC FT for 'r'
0  Ident    0.0                  :  2.1
1  Return   5.1                  :  0.0
2  Ident    0.1                  :  4.1 3.5
3  IApply   =2 =17 4.1 =0 2.2    :  5.2                (6)
4  DNum0    2.1                  :  3.3
5  Apply    =11 3.1              :  1.1
6  End      0
*17                                                    DDC FT for 'wvr' cond.
0  Ident    0.0                  :  2.1
1  Return   7.1                  :  0.0
2  Ident    0.1                  :  3.1 8.2
3  Ident    2.1                  :  4.2 6.2
4  Apply    =11 3.1              :  5.1
5  Mod      4.1 6.1              :  7.1
6  Con      =2 3.2               :  5.2
7  Eq       5.1 8.1              :  1.1
8  Con      =0 2.2               :  7.2
9  End      0
*18                                                    DDC FT for a.p.
0  Ident    0.0                  :  2.2
1  Return   2.1                  :  0.0
2  Apply    =8 0.1               :  1.1
3  End      0
*19                                                    List
```

Figure A5.1. A data flow program which implements the program of Figure 6.7.

Figure A5.2 shows the output produced by the data flow graph interpreter when the program of Figure A5.1 is executed.

```
Data Flow Graph Interpreter
Do you want tracing?  n
Which Fn to run ?  0
Result at output 1 = 633
```

Figure A5.2. Result of execution of program of Figure A5.1.

REFERENCES

[Abd76]
Abdali, S.K. A lambda calculus model of programming languages (I and II). *Journal of Computer Languages*, Vol. 1 (1976).

[Ack78]
Ackermann, W.B. A Structure Processing Facility for Data Flow Computers. {CSG Memo 156, MIT Lab. for Computer Science, Jan 1978}.

[AckD79]
Ackermann, W.B. and Dennis, J.B. VAL – A Value Oriented Algorithmic Language. {MIT/LCS/TR-218, MIT, June 1979}.

[Ack82]
Ackermann, W.B. Data flow languages. *Computer*, Vol. 15, No. 2 (Feb. 1982).

[ACM82]
Computer Architecture News (Apr 1982 and Jun. 1983).

[Ada71]
Adams, D.A. A model for parallel computations. In *Parallel Processor Systems, Technologies and Application*, L.C. Hobbs et al (eds.) (Spartan Books, 1971).

[AllO79]
Allan, S.J. and Oldehoeft, A.E. A flow analysis procedure for the translation of high level languages to a data flow language. In *Proc. 1979 Intl. Conf. on Parallel Processing, IEEE, Aug 1979*, O.N. Garcia (ed.).

[AllO80]
Allan, S.J. and Oldehoeft, A.E. Loop decomposition in the translation of sequential languages to dataflow languages. In *Proc. 1980 Intl. Conf. on Parallel Processing, IEEE, 1980.*

[Ama82]
Amamiya, M. et al A list–processing oriented data flow machine. In *Proc. National Computer Conf., 1982.*

[AmaHM82]
Amamiya, M., Hasegawa, R. and Mikami, H. List processing with a data–flow machine. In *Lecture Notes in Computer Science, vol. 147, 1982.*

[ArvG78]
Arvind and Gostelow, K.P. Some relationships between asynchronous interpreters of a data flow language. In *Formal Descriptions of Programming Concepts*, E.J. Neuhold (ed.) (North–Holland, 1978).

[Arv80]
Arvind. Decomposing a program for multiple processor systems.. In *Proc. 1980 Intl. Conf. on Parallel Processing, IEEE, 1980.*

[ArvT81]
Arvind and Thomas, R.E. I–structures: An Efficient Data Type for Functional Languages. {MIT/LCS/TM-210, MIT, Oct. 1981}.

[ArvGP78]
Arvind, Gostelow, K.P. and Plouffe, W. An Aasynchronous Programming Language and Computing Machine. {Uni. California at Irvine, TR114A, Dec. 1978}.

[ArvA82]
Arvind and Agerwala, T. Data flow systems. *Computer*, Vol. 15, No. 2 (Feb 1982).

[ArvG82]
Arvind and Gostelow, K.P. The U–interpreter. *Computer*, Vol. 15, No. 2 (Feb. 1982).

[ArvI83]
Arvind and Ianucci, R.A. A critique of multiprocessing von Neumann style. *Computer Architecture News* (June 1983).

[AshW76]
Ashcroft, E.A. and Wadge, W.W. Lucid–a formal system for writing and proving programs. *SIAM J. Comp.*, Vol. 5, No. 3 (Sep 1976).

[AshW77a]
Ashcroft, E.A. and Wadge, W.W. Lucid, a non–procedural language with iteration. *CACM*, Vol. 20., No. 7 (Jul. 1977).

[AshW77b]
Ashcroft, E.A. and Wadge, W.W. Intermittent Assertion Proofs in Lucid. In *Information Processing 77* (North–Holland, 1977).

[AshW78]
Ashcroft, E.A. and Wadge, W.W. Clauses–scope structures and defined functions in Lucid. In *Proc. 5th. Annual ACM Symposium on Principles of Programming Languages* (ACM, 1978).

[AshW79a]
Ashcroft, E.A. and Wadge, W.W. A Logical Programming Language. {TR CS-79-20, Dept. of Computer Science, Uni. of Waterloo, 8 June 1979}.

[AshW79b]
Ashcroft, E.A. and Wadge, W.W. Structured Lucid. {TR CS-79-21, Dept. of Computer Science, Uni. of Waterloo, June 1979}.

[Bac78]
Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of papers. *CACM*, Vol. 21, No. 8 (Aug. 1978).

[Ber75]
Berkling, K. Reduction languages for reduction machines. In *Proc. 2nd Int. Symp. Comp. Arch.* (IEEE, 1975).

[BroM79]
Brock,J.D. and Montz L.B. Translation and optimization of data flow programs. In *Proc. 1979 Intl. Conf. on Parallel Processing, IEEE, 1979*.

[Bur75]
Burge, W.H. *Recursive Programing Techniques.* (Addison–Wesley, 1975).

[BurMS81]
Burstall, R.M., MacQueen, D.B. and Sanella, D.T. HOPE: An Experimental Applicative Language. {CSR-62-80, Dept of Computer Science, Uni. Of Edinburgh, May 1980, rev. Feb. 1981}.

[CalDP82]
Caluwaerts, L.J., Debacker, J. and Peperstraete, J.A. A data flow architecture with a paged memory system. *Computer Architecure News* (Apr. 1982).

[CalDP83]
Caluwaerts, L.J., Debacker, J. and Peperstraete, J.A. Implementing streams on a data flow computer system with paged memory. *Computer Architecure News* (June 1983).

[Car76]
Cargill, T.A. Deterministic operational semantics for Lucid. {TR CS-76-19, Dept. of Computer Science, Uni. of Waterloo, June 1976}.

[CatG80]
Catto, A.J. and Gurd, J.R. Nondeterministic dataflow graphs. In *Information Processing 80* (North–Holland, 1980).

[Cha71]
Chamberlin, D.D. The single assignment approach to parallel processing. In *Proc. AFIPS FJCC, 1971.*

[ClaGMN80]
Clarke, T.J.W., Gladstone, P.J.S., Maclean, C.D. and Norman, A.C. SKIM-the S,K,I reduction machine. In *Proc. LISP-80 Conf., Stanford, Aug. 1980.*

[ComHS80]
Comte, D., Hifdi, N. and Syre, J.C. The data driven LAU multiprocessor system: results and perspectives. In *Information Processing 80* (North-Holland, 1980).

[Con63]
Conway, M.E. Design of a separable transition-diagram compiler. *CACM*, Vol. 6, No. 7 (July 1963).

[DarR81]
Darlington, J and Reeve, M. ALICE: a multiprocessor reduction machine for the parallel evaluation of applicative languages.. In *Proc. 1981 Conf. on Functional Programming Languages and Architecture, ACM, 1981.*

[Dar82]
Darlington, J. et al (eds) *Functional Programming and its Applications* (CUP, 1982).

[Dav78]
Davis, A.L. Data driven nets: a maximally concurrrent, procedural, parallel process representation for distributed control systems. {UUCS-78-108, Dept. of Computer Science, Uni. of Utah, July 1978}.

[Dav79]
Davis, A.L. A Data flow evaluation system based on recursive locality. In *AFIPS Conference Proceedings, vol. 48, 1979.*

[Den83]
Denbaum, C.A. *A Demand-Driven, Coroutine-Based Implementation of a Nonprocedural Language* {PhD Thesis, Dept. of Computer Science, Uni. of Iowa, 1983}.

[DenM83]
Denbaum, C.A. and Marlin, C.D. Coroutine-based implementation of nonprocedural programming languages. {GWU-IIST-83-10, George Washington Uni, Aug 1983}.

[Den74]
Dennis, J.B. First Version of a Data Flow Procedure Language. {MIT/LCS/TM-61, MIT, 1974}.

[DenFL74]
Dennis, J.B. Fosseen, J.B. and Linderman, J.P. Data flow schemas. In *Lecture Notes in Computer Science, vol. 5, 1974.*

[DenM75]
Dennis, J.B. and Misunas, D.P. A preliminary architecture for a basic data flow processor. In *Proc. 2nd Ann. Symp. Computer Architecture* (IEEE, 1975).

[Den80]
Dennis, J.B. Dataflow supercomputers. *IEEE Computer*, Vol. 13, No. 10 (Nov. 1980.).

[Den81]
Dennis, J.B. An operational semantics for a language with early completion data structures. In *Lecture Notes in Computer Science, vol. 107, 1981.*

[DenGT84]
Dennis, J.B., Gao, G-R. and Todd, K.W. Modeling the weather with a data flow supercomputer. *IEEE Trans. on Computers*, Vol. C-33, No. 7 (July 1984).

[Far77]
Farah, M. *Correct Compilation of a Useful Subset of Lucid* {Ph.D. Thesis, Dept. of Computer

Science, Uni. of Waterloo, Oct. 1977}.

[FarGT79]
Farrell, E.P., Ghani, N. and Treleaven, P.C. A concurrent computer architecture and a ring based implementation. In *Proc. 6th Annual Symp. Computer Architecture, IEEE, 1979.*

[Fau82]
Faustini, A.A. An operational semantics for pure data flow. In *Lecture Notes in Computer Science, vol. 140, July 82.*

[Fau83]
Faustini, A.A., Matthews, S.G. and Yaghi, A.AG. The pLucid Programming Manual. {Distributed Computing Proj. Rep. 4, Dept. of Comp. Sci, Uni. of Warwick, April 1983}.

[FreG83]
French, E. and Glaser, H. TUKI, a dataflow processor. *Computer Architecture News* (June 1983).

[FriW76]
Friedman, D.P. and Wise, S. The impact of applicative programming on multiprocessing. In *Proc. 1976 Intl. Conf. on Parallel Processing, IEEE, 1976.*

[FriW78]
Friedman, D.P. and Wise, S. Functional combination. *Computer Languages*, Vol. 3 (1978).

[Gaj81]
Gajski, D.D., Kuck, D.J. and Padua, D.A. Dependence driven computation. In *CompCon 81, IEEE, 1981.*

[Gel76]
Gelly, O. et al LAU system software. In *Proc. 1976 Intl. Conf. on Parallel Processing, IEEE, 1976.*

[Gla82]
Glasgow, J.I. *An Operational Semantics for Lucid and its Correctness* {Ph.D. Thesis, Uni. of Waterloo, 1982}.

[GurWG80]
Gurd, J. Watson, I. and Glauert, J. A Multilayered Data Flow Computer Architecture. {Draft TR, Uni. of Manchester, Mar 1980.}.

[GurGK81]
Gurd, J.R., Glauert, J.R.W. and Kirkham, C.C. Generation of dataflow graphical object code for the LAPSE programming language. In *Lecture Notes in Computer Science, vol.111, June 1981.*

[GurW83]
Gurd, J. and Watson, I. Preliminary evaluation of a prototype dataflow computer. In *IFIP 83* (North-Holland, 1983).

[HanG81]
Hankin, C.L. and Glaser, H.W. The data flow programming language CAJOLE. *SIGPLAN Notices*, Vol. 6, No. 7 (July 1981).

[Hen80]
Henderson, P. *Functional Programming: Application and Implementation* (Prentice-Hall, 1980).

[Hoa78]
Hoare, C.A.R. Communicating sequential processes. *CACM*, Vol. 21, No. 8 (Aug 1978).

[Hof78]
Hoffmann, C.M. Design and correctness of a compiler for a nonprocedural language. *Acta Informatica*, Vol. 9 (1978).

[Hof80]
Hoffmann, C.M. Semantic properties of Lucid's compute clause and its compilation. *Acta Informatica*, Vol. 13 (1980).

[Isa79]
Isaman, D.L. Data-Structuring Operations in Concurrent Computations. {MIT/LCS/TR-224, MIT, Oct 1979.}.

[Joh71]
Johnston, J.B. The contour model of block structured processes. *SIGPLAN Notices*, Vol. 7, No. 2 (Feb. 1971).

[JonKM85]
Jones, P.E.C., Kidman, B.P. and Morello, R. Code for generation from expressions in a data flow language. *Australian Computer Science Comm.*, Vol. 7, No. 1 (Feb 1985) {Proc. 8th Aust. Comp. Sci. Conf., Uni of Melboure, Feb. 1985.}.

[Kah74]
Kahn, G. The semantics of a simple language for parallel programming. In *Information Processing 74* (North-Holland, 1974).

[KahM77]
Kahn, G. and MacQueen, D.B. Coroutines and networks of parallel processes. In *Information Processing 77* (North-Holland, 1977).

[KarM66]
Karp, R.M. and Miller, R.E. Properties of a model for parallel computations. *SIAM J. Appl. Math.*, Vol. 14, No. 6 (Nov. 1966).

[KelLP78]
Keller, R.M. Lindstrom, G. and Patil, S. An architecture for a loosely coupled parallel processor. {UUCS-78-105, Dept. of Computer Science, Uni. of Utah, Oct.78.}.

[KelLP79]
Keller, R.M. Lindstrom, G. and Patil, S. A loosely coupled applicative multiprocessing system. In *AFIPS Conference Proceedings, vol. 48, 1979.*

[KisYK83]
Kishi, M., Yasuhara, H. and Kawamura Y. DDDP: a distributed data driven processor. *Computer Architecture News* (June 1983).

[Kli72]
Klinkhamer, J.F. A definitional language. In *Online 72* {Proc. Intl. Conf. on Online Interactive Computing}.

[Kos73]
Kosinski, P.R. A Data Flow Programming Language. {IBM Thomas J. Watson Research Center, RC 4264, March 1973}.

[Lan65]
Landin, P.J. A correspondence between ALGOL and Church's Lambda notation. *CACM*, Vol. 8, No. 2 (Feb 1965).

[Lan66a]
Landin, P.J. The next 700 programming languages. *CACM*, Vol. 9, No. 3 (Mar. 1966).

[Lan66b]
Landin, P.J. A lambda calculus approach. In *Advances in Programming and Non-Numerical Computers*, L. Fox (ed.) (Pergamon Press, 1966).

[Man74]
Manna, Z. *Mathematical Theory of Computation* (McGraw Hill, 1974).

[Mar80]
Marlin, C.D. *Coroutines: A Programming Methodology, a Language Design and an Implementation* {Lecture Notes in Computer Science, vol 95, 1980}.

[MauO83]
Maurer, P.M. and Oldehoeft, A.E. The use of combinators in translating a purely functional language to low-level data-flow graphs. *Computer Languages*, Vol. 8, No. 1 (1983).

[Mic68]
Michie D. Memo functions and machine learning. *Nature*, Vol. 218 (April 1968).

[Mis77]
Misunas, D.P. Report on the Workshop on Data Flow Computer and Program Organization. {MIT/LCS/TM-92, MIT, Nov 1977}.

[Mis78]
Misunas, D.P. A Computer Architecture for Data Flow Computation. {MIT/LCS/TM-100, MIT, March 1978}.

[OldTRZ77]
Oldehoeft, A.E., Thoreson, S.A., Retnadhas, C. and Zingg, R.J. The Design of a Software Simulator for a Data Flow Computer. {TR 77-2, Iowa State University}.

[Pil83]
Pilgram, P.T. Translating Lucid Data Flow into Message Passing Actors. {Distributed Computing Project Report 5, Dept.of Computing Science, Uni. of Warwick., Dec. 1983. (PhD thesis)}.

[Rei78]
Reid, T.F.. *The Specification of a Decompilation Methodology for Data Flow Schema* {Ph.D. Thesis, Dept. of Computer Science University of SW Louisiana, Nov. 1978}.

[ReqM83]
Requa, J.E. and McGraw, J.R. The piecewise data flow architecture:architectural concepts. *IEEE Trans. on Computers*, Vol. C–32, No. 5 (May 1983).

[Rod69]
Rodriguez, J.E. A Graph Model for Parallel Computation. {TR MAC-TR-64, Project MAC, MIT, Sep 1969}.

[Sri81]
Srini, V.P. An architecture for extended abstract data flow. {Proc. 8th Annual Symposium on Computer Architecture} *Computer Architecture News*, Vol. 9, No. 3 (May 1981).

[Syr82]
Syre, J.C. The data flow approach for MIMD multiprocessor systems. In *Parallel Processing Systems*, D. J. Evans (ed.) (CUP, 1982).

[Tan82]
Tanaka, H. et. al The preliminary research of data flow machine and data base machine as the basic architecture of fifth generation computer systems. In *Fifth Generation Computer Systems* (North-Holland, 1982).

[TesE68]
Tesler, L.G. and Enea, H.J. A language design for concurrent processes.. In *AFIPS Conference Proceedings, vol. 32, 1968*.

[Tho81]
Thomas, R.E. A Dataflow Architecture with Improved Asymptotic Performance. {MIT/LCS/TR-265, MIT, April 1981}.

[Tod81]
Todd, K.W. High Level VAL Constructs in a Static Data Flow Machine. {MIT/LCS/TR-262, MIT, June 1981}.

[TreH81]
Treleaven, P.C. and Hopkins, R.C. Decentralized computation. *Computer Architecture News* (May 1981).

[TreBH82]
Treleaven, P.C. Brownbridge, D.P. and Hopkins, R.P. Data driven and demand driven computer architecture. *Computing Surveys*, Vol. 14 (Mar. 1982).

[TreHR82]
Treleaven, P.C., Hopkins, R.P. and Rautenbach, P.W. Combining data flow and control flow computing. *Comp.J.*, Vol. 25, No. 2 (1982).

[Tur81]
Turner, D. The future of applicative programming. In *Lecture Notes in Computer Science, vol. 123, Oct. 1981*.

[Vee81]
Veen, A.H. Reconciling data flow machines and conventional languages. In *Lecture Notes in Computer Science, vol. 111, 1981*.

[Wad81]
Wadge, W.W. An extensional treatment of dataflow deadlock. *Theor. Comp. Sci.*, Vol. 13 (1981).

[WatG82]
Watson, I. and Gurd, J. A practical data flow computer. *Computer*, Vol. 15, No. 2 (Feb 1982).

[Weg71]
Wegner, P. Data structure models for programming languages. *SIGPLAN Notices*, Vol. 6, No. 2 (Feb. 1971).

[Wen81]
Wendelborn, A.L. Implementing a Lucid-like programming language. *Australian Computer Science Communications*, Vol. 3, No. 1 (May 1981).

[Wen82]
Wendelborn, A.L. A Data Flow Implementation of a Lucid-like Programming Language. {TR 82-06, Dept. of Computer Science, Uni. of Adelaide, July 1982}.

[Wen83]
Wendelborn, A.L. An input/output facility for LX and data flow. {Internal memorandum, Dept. of Computer Science, Uni. of Adelaide, July 1983}.

[Wen75]
Weng, K.S. Stream Oriented Computation in Recursive Data Flow Schemas. {MIT/LCS/TM-68, MIT, Oct. 1975}.

[Wen79]
Weng, K.S. An Abstract Implementation for a Generalized Data Flow Language. {MIT/LCS/TR-228, MIT, May 1979}.

[Wir76]
Wirth, N. *Algorithms + Data Structures = Programs* (Prentice-Hall, 1976.).