Translating Lucid Data Flow into Message Passing Actors

by

Paul Theo Pilgram

An inaugural dissertation
submitted for the degree of
Doctor of Philosophy.

Department of Computer Science
University of Warwick
Coventry
England

October 1983

Table of Contents

CHAPTER III: Imperative Programs and Message Passing

CHAPTER IV: The Translation

Author's long-term address:
Paul Th. Pilgram
Roonstr. 3
D – 4800 Bielefeld 1
West Germany

Acknowledgements

If it was common practice to have a thesis derated because it had been written under too ideal working conditions, I had a lot to fear.

My supervisor Bill Wadge deserves first mention. His informal guidance, his continuous attention and support, and the many discussions with him have paved my way throughout the Ph.D. course.

But my gratitude extends to all members of the Warwick Denotational Semantics Group: Pete Cameron, Toni Faustini, Forouzan Golshani, Steve Matthews, and Ali Yaghi. The atmosphere, there, of permanent debate and learning from each other was hard to better. The same can be said about the Warwick Department of Computer Science altogether, which has been a friendly place and a pleasant environment to work in.

Financial support by the DAAD (German Academic Exchange Service, Bonn) for a considerable period of the Ph.D. course is gratefully acknowledged.

Also my thanks to the authors of UNIX® (the famous trademark of Bell Laboratories). The production of this thesis has been helped a lot by UNIX®. UNIX®, YACC and C are also the basis for "our" (i.e. Ostrum's [Ost81]) experimental pLucid system, parts of which are re-used in this thesis.

Throughout my studies many friends and relatives have given me their moral support and lent me a patient ear. This has meant a lot to me, especially when the going was hard.

I wish to dedicate this thesis to my mother.

Declaration

The work described in this thesis, except where stated explicitly in the text, is the result of my own original research.

Chapter II mainly sums up the language Lucid, developed by E.A. Ashcroft and W.W. Wadge. That chapter and appendix A use some material from the pLucid manual [FMY83].

Further, this dissertation is not substantially the same as any that I have submitted for a degree or diploma or any other qualification at any other University. No part of this thesis has already been or is being concurrently submitted for any such degree, diploma or other qualification.

Paul Th. Pilgram.

Translating Lucid Data Flow into Message Passing Actors

P. Th. Pilgram

Department of Computer Science
University of Warwick
Coventry CV4 7AL
England

ABSTRACT

This thesis is the first translation of ***full*** Lucid into code for von Neumann machines ("***imperative code***"). It demonstrates that it is possible to produce efficient code even in the presence of advanced features such as "***currenting***", ***recursive functions*** or operators whose semantics favour ***concurrency***. Earlier compiled implementations stopped well short of this.

Lucid is a family of ***non-procedural*** programming languages, invented by Wadge and Ashcroft. Lucid is neither tied to any particular data algebra, nor to a particular implementation technique. However, ***Data Flow*** (with its variants) lends itself particularly well to the implementation of Lucid.

Message Passing Actors is an imperative programming technique which leaves scope for cooperating concurrency. This benefits hardware (multi–computers, "***transputers***") and software technology alike. In this thesis, LUX, a PASCAL–like language with Message Passing Actors, has been chosen as the target language.

It is shown that there is a subset of Lucid (a "***nucleus***") which has the ***same*** expressive capacity as full Lucid. The nucleus is easier to implement than full Lucid. As a prerequisite for the translation, a LUX actor equivalent is formulated for each operator of the nucleus, once and for all. The design of these operator–actors is strongly guided by the execution strategy of ***demand driven*** Data Flow ("***lazy evaluation***"). Their data storage is based on FIFO queues ("***pipelines***"). The actors operate concurrently, but they harmonise their actions by exchanging messages which follow an agreed ***protocol***.

The translation is carried out in successive stages. First the Lucid program is transformed to make it lie entirely within the nucleus. The program is then mapped into LUX, where each operator is represented by an operator–actor and the references to the variables are manifested in the environment setup of these actors. Finally, the LUX code is made more efficient by the application of a variety of analysis and optimisation methods.

Lucid programs can be analysed for various properties, and the resulting information can assist the code optimisation (while also revealing program errors). Particularly important among these program analyses is a queue length determination based on Wadge's ***Cycle Sum Test***.

Keywords: ***non–procedural languages, Lucid, recursive functions, cycle sum test, program transformation, data flow, lazy evaluation, message passing, concurrency, transputers, Occam.***

CHAPTER I: Introduction

1.1 Aims and Objectives

This thesis is the first translation of ***full*** Lucid [AsW80, AsW83] into code for von Neumann machines ("***imperative code***", ↑ 3.1). It demonstrates that it is possible to produce efficient code even in the presence of advanced features such as "***currenting***", ***recursive functions*** or operators whose semantics favours ***concurrency***. Earlier compiled implementations stopped well short of this. Up to now, Lucid had all the benefits inherent in non-procedural languages, but its implementations were lacking in ***efficiency*** and in means for ***concurrency***.

Let me explain the title of the thesis, its method of investigation, and then make some general remarks. Up-arrows ↑ will quote the sections where full detail can be found.

1.1.1 The Title

Lucid is a family of ***non-procedural*** programming languages, invented by W.W. Wadge and E.A. Ashcroft. Such languages make a significant contribution to the advancement of software technology. This thesis treats Lucid rather as a "given", so there is little need to point out its specific attractions (↑ 3.5). Every Lucid program consists only of ***assertions***; each assertion defines a variable or a function. Every Lucid variable symbolises an infinite sequence of data objects, called a "***history***".

Lucid is neither tied to any particular data algebra, nor to a particular implementation technique. However, **Data Flow** (with its variants) lends itself particularly well to the implementation of Lucid. Throughout this thesis, the term ***Data Flow*** ("**DF**", ↑ 2.5) comprises the **data driven** as well as the **demand driven** ("***lazy evaluation***" [HeM76, FrW76]) variant. The method presented in this thesis extends to Data Flow languages in general.

The syntax of Lucid has been revised a few times over the years, but the concepts behind Lucid have remained untouched. This thesis refers (↑ 2.1) essentially to the version described in the book on Lucid [AsW83, also FMY83]; this version is much more usable than earlier ones. Substantial programs have been written in this version of Lucid (e.g. a screen editor), and Lucid can thus no longer be called an academic plaything.

Message Passing Actors (↑ 3.2) is an imperative programming technique which leaves scope for cooperating concurrency. In this thesis, the target language is **LUX**, a PASCAL–like language with Message Passing Actors. LUX (↑ 3.4) has been designed so as to facilitate the translation into any given concurrent language. LUX contains, among others, a special message passing technique ("*exceptions*", ↑ 3.4.2) which supports control of concurrent computations without burdening program execution and without disturbing the program's overall design.

1.1.2 The Method

It is shown that there is a subset of Lucid (a "**nucleus**") which has the *same* expressive capacity as full Lucid. The nucleus is easier to implement than full Lucid. As a prerequisite for the translation, a LUX actor equivalent is formulated for each operator of the nucleus, once and for all (↑ 4.5 f). The design of these operator–actors is strongly guided by the execution strategy of *demand driven* DF. Their data storage is based on FIFO queues ("**pipelines**", ↑ 4.6.1). The actors operate concurrently, but they harmonise their actions by exchanging messages which follow an agreed *protocol* (↑ 4.2).

The translation is carried out in successive stages. First the Lucid program is transformed to make it lie entirely within the nucleus. Next, it is transliterated into *Graph Lucid*. In Graph Lucid, each operator is represented by a node, and directed arcs express the references to the variables. The graph is then mapped (↑ 4.3) into

LUX, where each node corresponds to an operator–actor and the arcs are manifested in the environment setup of these actors. Finally (↑ 6.0 ff), the LUX code is made more efficient by the application of a variety of analysis and optimisation methods.

Wadge and Ashcroft outline in their article [AsW77a] three approaches for implementing Lucid:

(1) translation into a conventional language,

(2) use of data driven Data Flow,

(3) use of a demand driven interpreter,

and, according to [AsW77a] only approach **(3)** is able to correctly compute the least fixed point of ***arbitrary*** Lucid programs. The first stage of the implementation proper is easy: the Graph Lucid program is re–interpreted as the "block diagram" of a multi–computer system, every Lucid node being bijected to a processing unit (= an ***actor***). Next, we have to decide how the actors operate (i.e. their internal behaviour) and cooperate (i.e. how information is passed between them). Our long–term perspective is to execute Lucid programs efficiently on available hardware. As a first step towards this aim, we furnish the actors with characteristics for which ***good code for conventional computers can be formulated***. The emerging multi–actor code is subsequently tuned for the target machine.

The method described in this thesis avoids the rigid commitment to any single approach, and is thus able to enjoy the advantages of ***all*** of them. In spite of belonging to group **(1)**, it does not hide its demand driven origins **(3)**, but it can even employ data driven techniques **(2)** where indicated. This flexibility can be achieved by picking the most suitable act in each case. Program analysis can lead to further advice which specialised act to choose (↑ 6.4, 6.6).

The data storage in DF can be arranged in, mainly, either of two ways: ***pipelines*** (= FIFO queues) or ***tagged*** store. In the tagged method, the data are stored and retrieved in any order; all the data are held in an associative store, with tags indicating the identity of each data item. The tagged method is clearly very un–restrictive, but it requires quite a sophisticated control mechanism. The behaviour of the tagged store differs widely from the one inherent in conventional computers; it would therefore be hard to establish a correspondence between the two. On the other hand, almost every pipeline can be handled by a few simple machine instructions. In pipeline DF, histories can only be evaluated in a restricted sequence. There are, unfortunately, Lucid programs which are computable in tagged DF but not in pipeline DF. Occasionally, pipeline DF can even cause wasteful computations. However, we choose pipeline DF as the main method, because of its greater machine affinity, and treat tagged DF only as an emergency choice. Anyway, a totally general tagged DF implementation requires the program to be held in a special internal representation (data dependency graph) for which corresponding conventional code can be found only in some lucky cases.

Conventional computers offer no abundance of ***processing power***, and mere small–scale ***concurrency*** is provided only at high cost. ***Data driven*** DF implies very high concurrency, but it has little concern for efficiency: it produces masses of computation results in the hope that some of them will eventually be of use. In our context, this would be suitable only in select cases. On the other hand, ***demand driven*** DF is efficient, and requires little or no concurrency; this is therefore our prime choice.

The translation generates by default code with ***high*** concurrency (one actor per node). Even before their translation, Lucid programs can be analysed for various properties, and the resulting information can assist the code optimisation, while also revealing program errors († 6.0 ff). Particularly important among these program analyses is a queue length determination based on Wadge's ***Cycle Sum Test*** ([Wad79], † 6.1). The optimisation can be directed to ***minimise*** or to ***maximise*** concurrency as far as reasonable.

1.1.3 Concurrency

In sequential implementations, operators evaluate their operand usually from left to right. If the left operand of an operator like OR happens to get into an endless computation, the OR will never yield its result, even if its right operand is TRUE. This is not in accord with the generally accepted mathematical meaning of OR. One would expect the following equations to hold:

```
   a      OR      b        =   b OR a    (commutativity)
  TRUE    OR UNDEFINED     =    TRUE
UNDEFINED OR    TRUE       =    TRUE
```

One can therefore say that only a concurrent OR (↑4.5.3) adheres to the mathematical definition.

A further argument in favour of concurrency comes from the hardware arena. In the pursuit of ever increasing computing power, hardware designers have turned their attention to concurrent machines (multi–computers, "***transputers***"), sharing the computing load among many arithmetic units. The traditional programming languages were deliberately designed around mono–processors, and it is very hard to extract chances for concurrent evaluation from such programs. Lucid is not committed to any particular degree of concurrency, be it high or low, and it leaves therefore more scope to progress in the computer field than many of the old favourites.

Concurrency combines curse with benefit. On most present–day computers, concurrency can be achieved (simulated) only at considerable cost; it must therefore be minimised and reserved for those cases where there is no way around it. Programmers have developed a sense for avoiding concurrency, even for managing without it altogether. There are, however, significant programming tasks which are most naturally solved in a concurrent manner (e.g. breadth–first evaluations).

far as reasonable.

1.1.3 Concurrency

In sequential implementations, operators evaluate their operand usually from left to right. If the left operand of an operator like OR happens to get into an endless computation, the OR will never yield its result, even if its right operand is TRUE. This is not in accord with the generally accepted mathematical meaning of OR. One would expect the following equations to hold:

```
    a      OR      b          =   b OR a     (commutativity)
  TRUE     OR UNDEFINED       =    TRUE
UNDEFINED  OR    TRUE         =    TRUE
```

One can therefore say that only a concurrent OR (↑4.5.3) adheres to the mathematical definition.

A further argument in favour of concurrency comes from the hardware arena. In the pursuit of ever increasing computing power, hardware designers have turned their attention to concurrent machines (multi–computers, "*transputers*"), sharing the computing load among many arithmetic units. The traditional programming languages were deliberately designed around mono–processors, and it is very hard to extract chances for concurrent evaluation from such programs. Lucid is not committed to any particular degree of concurrency, be it high or low, and it leaves therefore more scope to progress in the computer field than many of the old favourites.

Concurrency combines curse with benefit. On most present–day computers, concurrency can be achieved (simulated) only at considerable cost; it must therefore be minimised and reserved for those cases where there is no way around it. Programmers have developed a sense for avoiding concurrency, even for managing without it altogether. There are, however, significant programming tasks which are most naturally solved in a concurrent manner (e.g. breadth–first evaluations).

Software technology would therefore benefit if concurrency no longer had to be circumvented at all cost. (A reasonable compromise would be to annotate each instance where an operator ***can*** be, but does not ***need*** to be, executed concurrently. Operator variants $\boxed{\text{ANDALSO}}$ and $\boxed{\text{ORELSE}}$ can indicate those cases where concurrency is dispensable.)

This thesis uses **concurrency load** to mean the number of actors which are at a particular moment ready to execute (i.e. actors which are not "***hung***" waiting for inputs). An excessive number of hung actors indicates often a design deficiency. This number can be reduced by combining particular actors into one. In a well tuned computer system the concurrency load is usually roughly equal to its number of CPUs.

Concurrent programs are executed non–deterministically. given the total state of a concurrent machine, one can generally not predict its next state with certainty. (In the absence of a system–wide universal time it would even be impossible to ***determine*** the total state [Lam78].) However, Lucid is a ***functional*** programming language; all its operators are such that the computation result of any Lucid program depends only on the program inputs, without any effect from the order of evaluation, i.e. it *is* deterministic.

1.1.4 Efficiency

The most heard objection against functional programming languages is their alleged inherent inefficiency. This thesis (like others before it) provides ample evidence that Lucid can be lifted to any level of efficiency: it all depends on the amount of optimisation. The conventional programming languages, on the other hand, are tailored for von Neumann monoprocessors, and a great effort is required to make them run efficiently on a machine with high concurrency; denotational programming languages (like Lucid) are superior in this respect.

1.2 Survey of Previous Work by Others

Quite a number of people have put a lot of effort into implementing Lucid. Some of these implementations were never completed, many were of unsatisfactory efficiency or covered only a subset of Lucid. In its early versions, Lucid was neither very powerful nor practical in actual use, and this hindered its wider acceptance. The syntax of Lucid has changed considerably over the years, and Lucid has now become quite a respectable language. Exotic constructs were abolished (e.g. function freezing) and useful ones were added. All this depreciated the older implementations, since they refer now to defunct languages. This problem (and the whole problem of language implementation) has been greatly alleviated with the invention of compiler–compilers, where a syntax change is so easily put into effect.

The following four versions of Lucid mark its main development stages:

- Basic Lucid ("BL", no connection to the language BASIC) is the oldest published version [AsW77a, AsW76]. It has assertions merely for variables, nested iteration is achieved by means of the intrinsic function "latest", but there are *no* user defined functions.
- We use the name Clause Lucid ("CL") [AsW77b] for a revised and extended version of BL; it has a block structure (*clauses*) in four variants which provides a non–procedural counterpart for procedures and functions, with and without iteration.
- Structured Lucid ("SL") [AsW80], based on USWIM [AsW79a], replaces CL's unwieldy clause variants by classing global variables as either elementary or non–elementary, its uniform $\boxed{\text{valof} \ldots \text{end}}$ *phrases* provide improved block structuring.
- Lucid 83 [FMY83, AsW83] puts $\boxed{\text{where}}$ clauses (ISWIM [Lan66]) in place of SL's $\boxed{\text{valof}}$ phrases, and a new technique called "currenting" removes the need for elementary variables.

CL, SL and Lucid 83 are of comparable expressive power. In terms of these versions of Lucid, and in terms of the implementation methods **(1)**, **(2)** and **(3)** (↑ 1.1.2), here are some stages in the development of Lucid implementations:

(a) M.D. May's BL interpreter, with arrays, tagged DF, written in BCPL (Warwick, around 1974), incomplete,

(b) Cargill's [Car76] BL interpreter, tagged DF,

(c) Wadge's BL interpreter, written in FORTRAN (around 1976), incomplete,

(d) Farah [Far77], formally compiling ***restricted*** CL into ALGOL,

(e) Hoffmann [Hof78], compiling ***restricted*** BL into ALGOL 60, written starting 1974,

(f) Gardin's [Gar78] CL interpreter, written in recursive FORTRAN/ALGOL, difficulties with portability,

(g) Bush [Bus79], data driven execution of CL on a DF machine,

(h) Wendelborn [Wen80 (Wen81)], compiling Lucid–W (***restricted*** CL) into Wirth's PL/0,

(i) Wendelborn's [Wen80, Wen82] data driven Data Flow interpreter for Lucid–W,

(j) Ostrum's [Ost81] "Luthid" interpreter (SL, written in C),

(k) Finch [Fin81], study of translating SL into Message Passing [Koc80],

(m) Sargeant [Sar82], demand driven execution of SL on a DF machine,

(n) Faustini's refinement of Ostrum's system (Lucid 83, "pLucid" [FMY83]),

(p) Denbaum [Den83], compiling ANPL (=CL) into the coroutine language ACL, tagged DF,

(q) This thesis, compiling Lucid 83 into the Message Passing language LUX, pipeline DF,

(r) Yaghi [Yag83], study of translating Lucid 83 into modal logic.

Each of these implementations is, in its way, a valuable contribution to functional programming and Lucid, but space considerations keep us from discussing each of them in the deserved length. The achievement of many of them lies in an area undisputed by this thesis anyway. For example, the formal studies underpinned that Lucid is indeed a formal system for proving program correctness: **(d)**, **(k)**, [AsW76], [Fau82], **(r)**. On the other hand, the early proper implementations gave people a means to gather "hands on" experience with the language, if nothing else. New implementations profited from their predecessors' achievements and mistakes.

All Lucid implementations comprise inevitably a front end which translates given Lucid programs into an internal representation where extraneous detail has been eliminated; this front end may be a UNIX® ***filter***. This filter consists of a lexical and a syntax analyser, two well known techniques of little new scientific challenge. The differences between CL, SL and Lucid 83 are largely neutralised in the output of this filter, so that, from this point on, we need no longer distinguish between the Lucid versions. The filter output is essentially a ***directed graph*** equivalent to the original Lucid program: every operator is mapped into a ***node***, and ***arcs*** express the way in which node inports "feed" from node outports. The ***direction*** of the arcs is the direction in which the computation results flow, and the arcs are ***labelled*** by identifiers of Lucid variables. The filter output can appropriately be called **Graph Lucid** (↑ 2.2). The machine internal ***representation*** of the graph is usually tailored for the subsequent stages (either forward or back pointers).

The remaining stages reflect the chosen implementation technique, and are therefore very dissimilar. Data driven Data Flow is almost impossible to implement without purpose made hardware, whereas demand driven Data Flow is the method commonly used in Lucid interpreters. The Lucid graph serves, in both cases, to direct the initiation of computing action.

Data driven DF **(2)**: whenever data become available, the graph indicates which further computations could benefit from these data. Still, the strategy for injecting data into the computation decides which Lucid programs are computable. It was mainly the need for special hardware which kept **(i)** from progressing beyond the paper study. A Lucid compiler **(1)** can occasionally employ techniques akin to data driven DF. Genuine data driven hardware was employed by **(g)** and **(m)**, they found that Lucid execution on such a machine requires not only abundant computing power but also abundant store.

Demand driven DF **(3)**: whenever data are requested, the graph indicates which other data are prerequired before the request can be fulfilled. Its prudent avoidance of waste and its easy sequential execution made demand driven DF the method used in all known Lucid *interpreters* (**(a)**, **(b)**, **(c)**, **(f)**, **(j)**, **(n)**). Such interpreters are ideally suited to using a *tagged* store, whereby they may even correctly execute *arbitrary* Lucid programs.

The *compiling* implementations, the type **(1)**, are here of greatest interest, since this thesis **(q)** is most closely related to them. These implementations use the graph less directly. They analyze it for various properties (↑6.1, 6.6), and use this information to generate code for the given machine. Many properties can be found only by such a *global* analysis. – Most code generators model the action of an interpreter, like **(j)**. They produce a linear sequence of instructions by "tree walking" the Lucid graph: whenever a new node is encountered they generate equivalent code. Compared to an interpreter, the compilation can anticipate some of the *administration*, once and for all.

Most of the older compiling implementations (viz. **(d)**, **(e)** and **(h)**) manage only to compile a severely restricted Lucid into imperative code. The problems in compiling full Lucid arise, since it is impossible to tell *in general* which parts of a history must be retained for succeeding computations. Wendelborn, for example, resolves this by permitting only single application of the $\boxed{\text{NEXT}}$ operator [Wen80], or by requiring assisting information [Wen82] (the programmer has to state the maximum buffer length). The former restricts the expressive power of Lucid quite severely, and the latter is rather against the spirit of Lucid.

We must analyze Denbaum's thesis **(p)** a bit more deeply, since its aims and achievements are harder to distinguish from those of *this* thesis; after all, **(p)** as well as **(q)** produce code in a "concurrent language". Denbaum claims even to implement totally unrestricted Lucid. However, **(p)** provides no control mechanism for concurrent operators (e.g. concurrent $\boxed{\text{OR}}$), and the target language ACL treats coroutines merely as a programming technique; its concurrency load is always 1, from which a multi–computer would hardly benefit. By contrast, both are clearly provided in **(q)**; its execution control mechanism, concurrent or not, is even rather central. (Efficiency is also largely neglected in **(p)**. No hints are provided how to evolve the method of **(p)** into a serious system.)

Already Farah [Far80] and Finch **(k)** point out the relevance of concurrency to Lucid implementations, and they see that concurrency is not easy to tackle. But **(q)** is the first to describe a technique for handling concurrent operators, and to achieve a concurrency load greater than 1.

1.3 The Notation Used

This thesis follows a rather informal style; it contains no high powered mathematics or elaborate proofs. An attempt has been made to illustrate every explanation with at least one example. All diagrams are placed in the text right where they are used, which makes the reading easier. Figures have a box drawn around them if they represent programs or excerpts from programs (in whatever language).

Further conventions throughout this thesis:

- Objects are printed in **bold** in their definition. In all these cases there is therefore no point in searching further up for a better definition. Bold printing is also used in introductions for highlighting very central terms. One-letter identifiers are usually emboldened in explanations to make them stand out.
- Objects are printed in a $\boxed{\text{box}}$ if they refer to objects from a program. Where appropriate, boxing is combined with bolding. Single-letter identifiers are usually not boxed but printed in bold. Boxing had to be omitted in drawings (due to problems in the printer software).
- *Italics* are used to give words a slight stress within the text, and also for quoting mathematical expressions (e.g. variables).
- The up-arrow (↑ ..) hints at chapters or figures where further detail can be found.

Various brackets are used in their habitual meaning:

[	]	bibliography references,
(	)	function arguments, subscription or just comments,
{	}	sets,
<	>	sequences, or BNF entities,
"	"	genuine quotations, or "weird" ways of putting things.

Simple conventions apply to identifiers in programs:

variables	are written in lower case,
keywords	are written in upper case (except in PASCAL programs, where this violates the standard),
procedure names	are written in lower case but with the initial in capitals.

Page numbers are printed in the top corner of every page, whereas the current section number is quoted in the bottom corner.

CHAPTER II: Lucid and Data Flow

2.1 The Lucid Syntax

This section describes the version of Lucid used in this thesis. This version is essentially the same as the subject of [FMY83, AsW83], only embellishments (e.g. lists and strings) have been omitted for the sake of clarity.

The programming language Lucid has little substance in common with languages like BASIC or PASCAL. The syntax of many programming languages resembles mathematical notation, but Lucid programs go much further: every Lucid program makes mathematical sense, it is the definition of the computation result written in mathematical notation. Just the same, Lucid is not difficult to grasp, knowledge of heavy maths is not required for understanding Lucid, instead, most of Lucid is clear once a few facts are understood.

Lucid is best understood as the combination of two things:

- the Lucid syntax (the notation for Lucid programs) and
- the Lucid algebra (the objects symbolised by Lucid variables, and also the operators on them).

We will first introduce the Lucid syntax rather informally, then the Lucid algebra (↑ 2.3), and show eventually how the two are brought together in the formulation of relevant computations. The Lucid syntax is described more formally in appendix A. The ultimate and authoritative description of Lucid is found in [AsW83] and [FMY83].

2.1.1 Definitions (Assertions)

The syntax of Lucid comprises only few constructs, which makes it very easy to learn. A typical Lucid program, the computation of the running average of **x**, looks as follows:

```
sum / n   WHERE
            n   =  1 FBY n+1 ;
            sum =  x + (0 FBY sum) ;
          END
```

Between the keywords `WHERE` and `END` you see two lines of text, the one going `n = ... ;` the other `sum = ... ;`. Both are ***simple definitions***, and exemplify as such the most central construct of Lucid.

> Every **definition** states that the object on the left hand side of "=" is forever identical to the object on the right hand side, the **definiens**.

Definitions (also called **assertions**) can be either simple definitions or function definitions (* 2.1.4). The left hand side of a **simple definition** is just an identifier, the name of a Lucid **variable** being defined; the right hand side is an expression telling what is symbolised by the variable. The definition causes the lhs and the rhs to be totally equivalent so that, in expressions, the reference to a variable may be replaced by its definiens, without any effect on the computation result.

Adjacent definitions may be swapped, i.e. it is irrelevant in which sequence definitions are written. There must never be more than one definition for the same variable.

These rules highlight that definitions are quite unlike the assignments in imperative languages. Every definition states the nature of an object, and it is valid once and for all, just as in mathematics.

2.1.2 Expressions

The rhs of every definition is an **expression**, for example `1 FBY n+1`. Indeed, every Lucid **program** has the form of an expression. The Lucid rules for expressions are quite like those rules in most higher programming languages. An expression consists in the simplest case of a constant or of a variable. A **constant** is either an integer, or one of the special keywords `TRUE`, `FALSE` or `ERROR`. A variable can be denoted by any

identifier (a letter followed by any number of letters or digits), for example c3po or n. Certain sequences of letters are reserved as **keywords**, and are therefore not eligible as identifiers; they are:

```
AND ASA CURRENT ELSE END EQ ERROR FALSE FBY FI FIRST GE GT
IF IS LE LT MOD NE NEXT NOT OR THEN TRUE UPON WHERE WVR
```

(EQ NE LE LT GE GT are the ***relational operators*** for comparing data, AND OR NOT are the Boolean operators, MOD is the division remainder, ASA FBY FIRST NEXT UPON WVR are special functions of the Lucid algebra (* 2.3), while IF THEN ELSE FI WHERE END and IS CURRENT have other uses.)

Complicated expressions can be built out of simpler ones: a ***prefix operator*** (− NOT FIRST NEXT) can be put in front of an expression, or an ***infix operator*** can be placed between two expressions, the outcome is a bigger expression in either case. Ambiguities can be resolved by enclosing an expression in brackets before building such a bigger expression. In most cases, however, brackets are unnecessary since a **precedence** is defined among the operators.

```
(strongest binding)
10   FIRST  NEXT
 9   *   /   MOD
 8   +   −
 7   EQ  NE  LT  LE  GT  GE  <   <=   >   >=
 6   NOT
 5   AND
 4   OR
 3   FBY
 2   ASA  UPON  WVR
 1   WHERE
(weakest binding)
```

The precedence is the "relative binding force" of the operators. What is meant is this: any operator with a high precedence (strong binding force) can "grab hold of its operands" before an operator with a lower precedence (weaker binding force) can try. In the expression 1 FBY n + 1 the variable **n** has an infix operator on either side, and we can read from the precedence table that + binds more strongly than FBY. The + operator will therefore win over FBY in claiming **n** as operand, thus making the whole expression equivalent to `1 FBY (n + 1)`. The binding is ***left*** associative among operators of equal precedence, except for `FBY` where it is right associative, so that:

```
a - b - c - d  =  ((a - b) - c) - d
```

whereas

```
a FBY b FBY c FBY d  =  a FBY (b FBY (c FBY d))
```

Certain operators (viz. the comparison operators) do not associate at all, i.e. separating two such operators only by an operand makes no sense in Lucid:

```
0 < n  AND  n < 1000        // is a legal expression,
0 < n < 1000                // is incorrect.
```

`IF c THEN t ELSE e FI` is an **if–expression** with **c**, **t** and **e** being expressions; the condition operand **c** selects whether the result of the `IF` is taken from **t** or from **e** (↑ 2.3.3.1).

Expressions can also contain **function references**, such as `f(x)` or `ggt(i*i+3, k+j)`. Every function reference starts with the function identifier, followed by the actual parameters in brackets. Each actual parameter is an expression. A definition of the function (i.e. with the same identifier, ↑ 2.1.4) must be provided in a suitable position (scope rules: ↑ 2.1.5).

`WHERE` clauses are a further construct permitted in expressions, a construct so crucial to deserve its own section:

2.1.3 `WHERE` clauses

In our Lucid example program, a `WHERE` **clause** constituted the top level structure. This is perfectly legal, since `WHERE` clauses constitute expressions, and only expressions constitute Lucid programs. The BNF (↑ appendix A) of a `WHERE` clause is:

```
<expression>
  WHERE
    <currenting>    // any number of these
    <definition>    // any number of these
  END
```

Everything between the WHERE and its corresponding END is called the WHERE **body**, while the WHERE **expression** is the expression on the left of WHERE. Right after the keyword WHERE is the only place where **currentings** are permitted. A currenting (↑ appendix B) has the BNF:

`<variable> IS CURRENT <expression> ;`

and it defines the ***<variable>*** to be, in a special way, equal to the ***<expression>***; incidentally, this expression is evaluated ***outside*** the WHERE clause. Currenting is quite an involved matter, so that appendix B should be read only after completion of this entire chapter.

We are now able to present a program which contains all the syntactic features of Lucid:

```
s+1  ASA  s EQ t
  WHERE
    x         IS CURRENT  x-1      ;
    y         IS CURRENT  z-1      ;
    c         = 1                  ;
    s         = x FBY chop (s,t)   ;
    chop(a,b) = a MOD (b+c)        ;   // ↑ 2.1.4
    t         = y FBY chop (t,s)   ;
  END
```

Each definition or currenting in the WHERE body attaches a meaning to an identifier, be it a variable or a function. The WHERE expression (here: s+1 ASA s EQ t), and the expressions within the WHERE ***definitions***, will usually refer to identifiers (of variables and functions). In order to determine the identity of the variable or function, the compiler performs a search, first among the definitions in the WHERE body and then outward through the syntactic structures which enclose the WHERE clause. (There is none of the latter in our example.) If no match is found, variables are assumed to be **input variables**, **x** and **z** in our example, whereas for functions an error must be reported.

2.1.4 Function Definitions and UDFs

Lucid programmers can also define functions of their own design; such functions are called **UDFs**, **User Defined Functions**. (Mathematically speaking, all Lucid operators are "functions".) The latter example program contains a definition of the function chop, and there are two references to that function. A **function definition** looks rather the same as a simple definition, except that on the left of the "=" sign we have the function name, followed by the formal parameters, in brackets. For example:

```
chop(a,b) = a MOD (b+c) ;
```

This defines a UDF chop, of two parameters, to be forever identical to the expression on the right hand side (the definiens). The definition declares also the formal parameters **a** and **b**; formal parameters must never share the same identifier. Each formal parameter is bound to its corresponding actual parameter in the function reference. Global variables (i.e. variables which are not formal parameters) are permitted in the definiens, like **c** in the example. We illustrate the use of UDFs by studying the function reference in:

```
s = x FBY chop(s,t) ;
```

The definiens of chop has free variables (**a**, **b** and **c**), and the function reference makes sense only after all the free variables have been bound properly. For this purpose an outward search is conducted, through all the structures which syntactically enclose the definiens. The first enclosing structure is the function definition, and the variables **a** and **b** are defined there as formal parameters. Formal parameter **a** is in this case bound to actual parameter **s**, and **b** is bound to **t**. It is in this case possible simply to rename the formal parameters, there being no clash of identifiers, and to substitute (macro expand) the function reference, giving:

`s = x FBY s MOD (t+c);`

Variable **c** is still free; it is bound only in the next enclosing structure, the `WHERE` clause, where we find its definition `c = 1;`.

2.1.5 Environments and Scope Rules

All function definitions appear in `WHERE` bodies, and `WHERE` clauses can appear in the definiens of a function. Both constructs can thus be arbitrarily nested, and either construct declares variables (or functions or formal parameters) to which reference can be made from inside the construct. The rule for identifier look—up has just been described once for `WHERE` clauses and separately again for function definitions. The compiler, however, uses in reality one and the same mechanism for both look—ups. Each function definition and every `WHERE` clause constitutes an **environment**, and each environment gives a meaning to some particular identifiers. Environments form a hierarchy (a tree). The input variables are contributed by the outermost environment. If an environment gives a new meaning to an identifier, this has the effect of locally superseding (making inaccessible) any meaning which that identifier may have had outside that environment.

We can draw the environments as dotted lines into our example program. That program contains three environments: the environment around the function definition, defining **a** and **b**, one around `WHERE` clause (with the currenting half sticking out), defining **x**, **y**, **c**, **s**, `chop` and **t**, and the outermost environment, defining **x** and **z**. The superseding applies here only to **x**:

```
 .................................
: s+1  ASA  s EQ t               :
:   WHERE                        :
:     x          IS CURRENT      :  x-1 ;
:     y          IS CURRENT      :  x-1 ;
:     c          = 1             :      ;
:     s          = x FBY chop (s,t)  :  ;
:            ....................:
:     chop : (a,b) = a MOD (b+c) :  :   ;
:          :.....................:  :
:     t          = y FBY chop (t,s)  :  ;
:...END..........................:
```

2.1.6 Program Transformations

Those readers who aim primarily to learn the language Lucid are advised to continue at section 2.3. The Lucid syntax comprises constructs which are "luxury" since they express, concisely, something that could also be expressed through the basic outfit, though at extra length. This luxury is perfectly justified in the programming language, since it helps to keep programs legible. However, when it comes to translating Lucid into some other code, a language is desirable with only a minimal spectrum of constructs, since obviously every construct requires its specific translation rule. The elimination of ***currenting*** is described in appendix B. This section presents methods for eliminating four things: identifier clashes, multi–operand expressions, global variables in functions, and multiple references to variables. All these eliminations can be done in separate compilation passes (e.g. UNIX® filters), in the sequence just mentioned. It does no harm if this pre–translation reduces the aesthetics of the program, since no human eye will read the program in this intermediate form anyway.

Unique Identifiers

Different environments may attach different meanings to the same identifier, by means of currentings, definitions or formal parameters. However, the later translation stages would benefit if all identifiers had a **unique** meaning. This state of affairs can be established by substituting identifiers by unique ones; this task is not hard since every program contains only finitely many identifiers. One might choose $\boxed{\mathtt{s}i}$ (i = *0, 1, 2,...*) as the substituting identifiers, though omitting the initial segment $\boxed{\mathtt{s0}}$ ... $\boxed{\mathtt{s}k}$ if the original program contains $\boxed{\mathtt{s}k}$ as identifier.

Monomeric Programs

Every definition has on the right of "=" an expression, and Lucid permits all expressions to contain *many* operators, by way of sub–expressions. The later stages or our translation, however, become particularly easy if only a single operator is permitted in any expression, and if every $\boxed{\mathtt{WHERE}}$ expression and every actual function parameter is required to be just a variable, if not a constant. In this way, the result of every operator can be associated with a variable ("*Operator*" is here meant in this most general sense which includes not only the prefix and infix operators, but also $\boxed{\mathtt{IF}}$ and all UDFs.) We call programs **monomeric** if they have been transformed in this way. Made monomeric, our example program looks as follows:

```
h0 WHERE
     x  IS CURRENT x-1 ,
     y  IS CURRENT z-1 ;
     c  =  1 ;
     chop(a,b) =  h5 WHERE
                       h6 = b + c ;
                       h5 = a MOD h6 ,
                      END ;
     s  =  x FBY h4 ;  h4 = chop (s,t) ;
     t  =  y FBY h3 ;  h3 = chop (t,s) ;
     h0 = h1 ASA h2 ;  h1 = s+1 ,  h2 = s EQ t ;
    END
```

The example demonstrates how easily the aim can be achieved: a definition for an auxiliary variable is inserted, where required, with the sub–expression serving as definiens. The auxiliary variables are named $\boxed{\mathtt{h}i}$ (i = 0, 1, 2, ..), though omitting i values which would clash with pre–existing identifiers;

If the definiens for a function needs to be broken into smaller expressions, a $\boxed{\text{WHERE}}$ clause must first be put around the definiens. We apply the rule that every **<expression>** can be blown up into:

$\boxed{\text{h WHERE h = <expression> ; END}}$

Global Variables in Functions

Global variables in functions are sometimes convenient for the programmer, but subsequent translation stages would come to grief with them. Global variables are easy to eliminate: they are simply added as extra actual and formal parameters both to the function definition and to each function reference (identifiers assumed to be unique). The respective lines in our example would change into:

```
s             = x FBY chop (s,t,c) ;
chop(a,b,c)   = a MOD (b+c)        ;
t             = y FBY chop (t,s,c) ;
```

$\boxed{\text{COPY}}$ definitions

We know that expressions can contain references to variables; this is the one and only way in which variables interconnect and eventually combine into the program. A variable may have more than one expression referring to it. No substantial program can do without such multiple references.

Since any operator may occur in a definition, every operator must be able to cope with multiple references. In a naive approach, one might implement each operator so that it can handle multiple references. Instead, we pretend that Lucid has an extra construct, namely the $\boxed{\text{COPY}}$ operator and the $\boxed{\text{COPY}}$ definition:

```
( <var> {,<var>} )  =  COPY ( <var> ) ;      // BNF

    (x, y, z)       =  COPY (a) ;            // example
```

$\boxed{\text{COPY}}$ is a unary multi–valued function; in the example, **x**, **y** and **z** refer to exactly the same variable **a**. Any number of <var>'s is permitted on the left hand side.

The entire problem of multiple references is now concentrated in the $\boxed{\text{COPY}}$ operator, all other operators will now have single references. (Note: The Lucid programmer is not allowed to use $\boxed{\text{COPY}}$ definitions.)

Lucid programs before and after all these transformations are shown in sections 4.3.3.1.

2.2 Graph Lucid

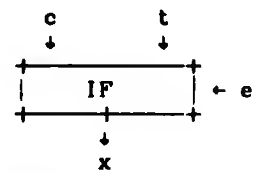
Before we turn to the Lucid algebra, let us use the occasion for introducing an entirely different program transformation, namely the one into **Graph Lucid**. Graph Lucid is not another programming language but only a different representation for Lucid programs; it serves mainly as a particularly suggestive illustration aid in our later explanations. The subject of section 2.1 might, in contrast, be called *equational* Lucid.

Given a Lucid program which has been conditioned according to appendix B and section 2.1.6, the translation into Graph Lucid is quite easy. In Graph Lucid, each operator is represented by a *node*, and *directed arcs* express the references to the variables. Let us study this in greater detail.

Every operator is mapped into a **node**. In our diagrams, nodes are drawn as boxes with the node type written inside. Every monomeric expression defines a result; correspondingly, every node has a point, called its **outport**, from where an arc springs. Generally, every operator has operands; correspondingly, every node has points, its **inports**, where arcs end. By convention, the outport is placed on the bottom line of the box, and inports are placed at the top or at either side. The sequence of the operands is reflected in the left–to–right sequence of the node inports; for example:

x = IF c THEN t ELSE e FI ;

may map into:



Matters are hardly different with COPY nodes; they differ only in so far as they have more than one outport. To limit the clutter in our diagrams, COPY nodes are symbolised by a plain letter **C**, and the node box is omitted.

Lucid programs express input and output implicitly, namely by means of the outermost environment (input) and by the overall result of the program (output). Graph Lucid requires one explicit READ node for each input variable, one WRITE node for the program result, and one CONSTANT node for each constant.

Expressions can contain references to variables and constants. Each reference is mapped, in Graph Lucid, into a **directed arc**. Every arc leads from an outport to an inport, i.e this is the direction of the arrow on the arc. Every arc can be unambiguously labelled with (the identifier of) a variable, often an *auxiliary* variable. We will occasionally speak of the **down**stream direction when we mean the arrowed direction of the arcs; **up**stream is the opposite, of course.

The translation of *UDFs* into Graph Lucid is described in section 4.3.2.2; until then, it is sufficient to know that every UDF is an operator, and the UDF parameters are its operands.

The beginning of section 4.3.3.1 shows how the example program Sieve would look when transformed into Graph Lucid. Labels **m_i** and **s_i** are used for auxiliary variables; the numbering is incidental, for the time being. – In the diagram, one COPY node (**s2**) is split up into three separate COPY nodes. Strictly speaking, this not perfectly legal; it has been used merely to keep the graph legible. – The letter **N** in the graph marks the point which corresponds to the variable **N** in the program. The graph on the left contains a **cycle**: we can run *down* the arcs from the PLUS node to FBY, then to COPY (C), and arrive again at PLUS. In Lucid programs, every cyclical definition needs to involve at least one variable; in the graph, the cycle can be broken at the point corresponding to this variable. This point is therefore called a **cutpoint**, and it is marked * in the graph. It coincides in our example with the variable **N**.

Any of our graphs is called a **net** if it has no open inports and outports (e.g. the left part of the Sieve graph), while a **subnet** is a graph with an open inport or outport (e.g. the right part). UDFs map into subnets, and the main program maps into a net.

2.3 The Lucid Algebra

2.3.1 Analogy

Lucid graphs are excellent for illustrating the Lucid concept. One can imagine the arcs were pipes, and there were plastic balls rushing down the pipes. Each ball contains an item of information, say, written on note paper. Instead of balls we speak of **datons**, and the information contained inside is called the **daton value**. Each pipe transports datons from a node outport to a node inport.

The nodes are machines, connected by the pipes in accordance with the program. The outports and inports resemble sockets with pipes attached. A node can check each of its inports whether it is *filled*, i.e. whether a daton is ready to be consumed. When given a daton at an inport, the node can take the daton, inspect its daton value, and take the appropriate action. The node produces datons with suitable value, and feeds them into the outport pipe.

Let us take for example the ADD node. It has two inports and one outport. Whenever each inport is filled, the node removes both datons, computes the sum of their values, and feeds a daton with the sum value into the outport.

On the other hand, the COPY node has one inport and ***at least*** one outport. Whenever the inport is filled, the node removes the daton, and feeds a copy of this daton into each of its outports.

Any network of nodes and pipes can be built up out of these components, and the computations take the form of daton processing and of pushing datons through the pipework. Looking at any point in the network of pipes and nodes, we see a ***stream of datons*** passing by (as long as the computation does not come to a halt). One can record the values of all the datons passing through a pipe, and one can say "this arc has this sequence of data associated".

If the program runs forever, it should compute an ***infinite*** sequence of data. Of course, only ***finitely*** many datons can be computed in finite time.

The analogy of the plastic balls has its limitations, it is merely meant as a rough guide. (It modelled the ***data driven*** version of ***pipeline*** Data Flow, ^ 2.5. We use the UNIX® term "**pipeline**" for FIFO queues in general.) Datons are in reality mere conceptual objects, and they can be produced and consumed without regard to any conservation law, as the description of the ADD and COPY nodes showed.

2.3.2 Datons and Histories

Datons are conceptual data ***particles***, whereas in conventional programming languages a data item is a mere contents of a storage cell. We confine the daton values to integers, TRUE or FALSE, or ERROR. Lucid allows, in principle, a much wider range of data, but the full generality would distract from the important points of this thesis.

We know that every variable of the Lucid program maps into an arc in the graph, and that every arc has a sequence (finite or infinite) of data associated. We call a finite or infinite sequence of data a **history**. Taken together, ***every Lucid variable has***

a history associated. Here are a few examples of histories:

```
index   = < 0, 1, 2, 3,   4,   5,   6,   7,  8,  9,  10, ... >
squares = < 0, 1, 4, 9,  16,  25,  36,  49, 64, 81, 100, ... >
primes  = < 2, 3, 5, 7,  11,  13,  17,  19, 23, 29,  31, ... >
chance  = < 46, -5, 0, 1537, 400, -34, -34, 1, 147,     ... >
```

(Warning: the sequence notation is only an aid for our discussion, it is not Lucid syntax.) The variable index is indeed ***predefined*** with the history shown above, because of its great practical use (i.e. it is known to Lucid even if the user does not define it).

The datons are by convention numbered from 0 up. This "serial number" is called the **index** of the daton. The daton with the index 0 is the **initial daton** ("first" could be misleading). We denote an individual daton of a history by writing its index as a subscript after the name of the history. The Lucid variable index is special in that for each daton the value is exactly its index ($\omega = \{0, 1, 2, \ldots\}$):

$$\text{index}_i = i \quad \forall \; i \in \omega$$

2.3.3 The Operators

The **algebra** is the specification both of the data objects and of the operations on them. Indeed, ***histories*** are the only Lucid data objects; every variable has a history associated. The daton values have their own algebra; this algebra is employed to generate a good part of the Lucid algebra. Here are the two algebras:

- The algebra of the daton values: its data objects are the integers, TRUE, FALSE and ERROR, its operators are the conventional operators (viz.: + − * / MOD IF AND OR NOT LT LE GT GE EQ NE).
- The Lucid algebra: its data objects are infinite sequences of datons, its operators are the special Lucid operators (NEXT FBY FIRST UPON WVR ASA) as well as the ***pointwise extensions*** of the conventional operators.

We explain now the operators: first the extension ("*lucidisation*") of the conventional operators, then the special Lucid operators, starting with the very important $\boxed{\text{FBY}}$ and $\boxed{\text{NEXT}}$ and followed by the more exotic operators.

2.3.3.1 The Pointwise Operators

All conventional operators can be extended ***pointwise*** (= ***index–wise***); such extended operators are **pointwise operators**. This operator extension is defined as follows: given a conventional operator ψ and given two histories **a** and **b**, the history $(a\ {}^{\omega}\psi\ b)$ is obtained by applying ψ individually to the operand datons:

$$(a\ {}^{\omega}\psi\ b)_i \ = \ a_i\ \psi\ b_i \quad \forall\ i \in \omega.$$

For example, a Lucid program may contain the simple definition:

```
sum  =  a + b ;   // "+" is here ωψ
```

This corresponds to the following equalities for individual datons:

$$\text{sum}_i \ = \ a_i + b_i \quad \forall\ i \in \omega$$

This is indeed the Lucid $\boxed{\text{ADD}}$ operator described in the analogy, above. Lucid operators yield an $\boxed{\text{ERROR}}$ daton whenever a proper result is barred by an error in the computation (e.g. a division by 0 is attempted). This is the most elegant and safe way of drawing attention to meaningless computation results.

Here is another simple definition:

```
pleasure  =  IF cond THEN music ELSE plants FI ,
```

The operand $\boxed{\text{cond}}$ is Boolean, i.e. each of its datons is either $\boxed{\text{TRUE}}$ or $\boxed{\text{FALSE}}$. Index by index, each daton of history $\boxed{\text{pleasure}}$ is the corresponding $\boxed{\text{music}}$ daton if the corresponding $\boxed{\text{cond}}$ daton is $\boxed{\text{TRUE}}$, otherwise it is the $\boxed{\text{plants}}$ daton.

```
plants   = <Rose, Tulip, Lily, Fern, Poppy, Grass, Fig, Triffid, ... >
music    = <Bach, Elvis, Ella, Duke, Holst, Haydn, Weill, Cliff, ... >
cond     = <TRUE, FALSE, TRUE, TRUE, FALSE, FALSE, TRUE, FALSE,  ... >

pleasure = <Bach, Tulip, Ella, Duke, Poppy, Grass, Weill, Triffid, ... >
```

Two points about IF must be highlighted:

- The result of the IF is obtained by inspecting the datons of its three operands at exactly the same index positions as the result, nothing needs to be known about datons at earlier or later index positions. Such operators are called **pointwise**. (The operators introduced in the remainder of this section 2.3.3 are not pointwise.)
- Dependent on the daton in the cond operand either the daton of the THEN or the ELSE operand is chosen for the result history. This means also that the value of the other daton is ***ignored***; the effort for its evaluation, if any, has been in vain.

2.3.3.2 The FBY Operator

Suppose, we have to write a Lucid program which generates the following history (the sequence notation is not permitted in Lucid).

```
h  =  < 1,  2,  3,  4,  5,  6,  7,  8,  9, 10,
       11, 12, 13, 14, 15, 16, 17, 18, 19, ... >
```

A proper definition of **h** can be based on its two characteristics

- the history starts with a 1 and
- the history proceeds in incremental steps of +1 .

The variable **h** can be defined by a recursive simple definition using the FBY operator (FBY stands for "***followed by***"):

```
h  =      1        FBY       h + 1   ;
   //     ↑                  ↑↑↑↑↑
   //   start              successor
```

The result of FBY is the history produced by taking the initial daton from the left operand (start) and by inserting it ahead of the history of the right operand (successor).

Any expression can be put at start, not merely our constant history of infinitely many 1–datons. Only the initial daton of start matters, it constitutes the initial daton of the result.

Any expression can be put at successor, and it constitutes the result from the daton h_1 on. One effect is that, comparing daton by daton the FBY result with its successor operand, the latter is always ahead by a single daton. Note the reference to **h** in successor: the definition is recursive. The following diagram illustrates how **h** is generated:

```
1         = <1,  1, 1, 1, 1, 1, 1, 1, 1,  1,  1,  1,  1,  1,  1,  1, ... >
             |
h         =  |  <1, 2, 3, 4, 5, 6, 7, 8,  9, 10, 11, 12, 13, 14, 15, ... >
1         =  |  <1, 1, 1, 1, 1, 1, 1, 1,  1,  1,  1,  1,  1,  1,  1, ... >
             |   ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓
             |
h+1       =  |  <2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ... >
             ↓   ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓

1 FBY h+1 = <1,  2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ... >

h         = <1,  2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ... >
```

The exact definition of FBY is:

$$(a\ \text{FBY}\ b)_0 = a_0$$

$$(a\ \text{FBY}\ b)_{i+1} = b_i \quad \forall i \in \omega$$

The following UDF, CountT, demonstrates the combined use of IF and FBY. It yields a running count of TRUE datons (CountT is a filter)

```
CountT (x)  =  s
       WHERE
        s   =  0 FBY IF x THEN s+1 ELSE s FI ;
       END ;
```

2.3.3.3 The FIRST Operator

FBY provides also a simple way to extract the initial daton from a history, deliberately discarding the rest of the history. This is achieved by:

```
s  =  fancy FBY s  ;
```

All datons of variable **s** are equal to the initial daton of fancy. It is also common to write:

```
s  =  FIRST fancy  ;
```

which means exactly the same, but is more convenient to write. — The exact definition of the FIRST operator is:

$$(\text{FIRST a})_i = a_0 \quad \forall\, i \in \omega$$

FIRST is semantically equivalent to the UDF:

```
First (a)  =  p  WHERE   p = a FBY p   END ;
```

2.3.3.4 The NEXT Operator

The NEXT operator is in a sense the inverse of FBY. The exact definition of NEXT is:

$$(\text{NEXT a})_i = a_{i+1} \quad \forall\, i \in \omega$$

Here is an example where NEXT is applied to a variable **h**:

```
n  =  NEXT h  ;
```

According to this definition, **n** is the history obtained by removing the initial daton from **h**. If **h** is defined as in the example above, we obtain:

$$\begin{aligned} \mathbf{n} \; = \; < & 2,\; 3,\; 4,\; 5,\; 6,\; 7,\; 8,\; 9,\; 10,\; 11, \\ & 12,\; 13,\; 14,\; 15,\; 16,\; 17,\; 18,\; 19,\; \ldots > \end{aligned}$$

Comparing, daton by daton, the NEXT result with its operand, the former is always ahead by a single daton. — NEXT is not the exact complement of FBY. The application of NEXT re-creates the successor operand of FBY, in other words:

```
c  =  NEXT ( a FBY b )  ;
```

gives c the history of b; a is irrecoverably ignored. c gets also the history of b in the following:

```
c  =  b FBY   NEXT b   ;
```

Let us study a simple example involving FIRST and NEXT:

```
deviation   =   NEXT ( r  -  FIRST r )  ;
```

We choose a random history for r and play the example through:

```
r             = <400, 970, 586, 946, 264, 640, 638, 117, 396, 743, 256, ... >
FIRST r       = <400, 400, 400, 400, 400, 400, 400, 400, 400, 400, 400, ... >
r - FIRST r   = <  0, 570, 186, 546,-136, 240, 238,-288,  -4, 343,-144, ... >
NEXT ( " )    =      <570, 186, 546,-136, 240, 238,-288,  -4, 343,-144, ... >
```

The following UDF, IndexT, is a more elaborate application of NEXT; it searches its Boolean operand k for a TRUE daton and returns its index position. Its integer operand i ($i \in \omega$) specifies which occurrence of TRUE is wanted: i = 0 requests the earliest occurrence.

```
IndexT (k, i) = IF   NOT k
                THEN IndexT (NEXT k, i) + 1
                ELSE IF   i > 0
                     THEN IndexT (NEXT k, i-1) + 1
                     ELSE 0
                FI   FI ;
```

2.3.3.6 The UPON Operator

The operators described in the remainder of this section 2.3.3 may look somewhat "artificial", but they are almost indispensable in any substantial Lucid program.

The UPON operator is of great use when we try to build a node which consumes datons (at an inport) at a slower pace than it produces them (at the outport). – Using the UDF CountT from above, the exact definition of UPON is:

$$(a \text{ UPON } k)_i = a_{(\text{CountT}(k)_i)} \quad \forall\, i \in \omega$$

The initial result daton of $\boxed{\text{a UPON k}}$ is a_0. Subsequently, if the operand **k** yields a $\boxed{\text{FALSE}}$ daton, the current daton of **a** is repeated once more; otherwise, the next daton of **a** is chosen for the result. – The $\boxed{\text{UPON}}$ operator is semantically equivalent to the following UDF:

```
Upon (a, k) =  a FBY Upon (p,  NEXT  k)
               WHERE p = IF    FIRST  k
                         THEN  NEXT   a
                         ELSE         a    FI ;
               END ;
```

As a typical use of $\boxed{\text{UPON}}$, here is the UDF $\boxed{\text{Mymerge}}$ which merges two histories **x** and **y**, under control of a Boolean $\boxed{\text{cond}}$, without losing any daton of **x** or **y**:

```
Mymerge (cond, x!, y!)  =
     IF cond THEN  x UPON      cond
             ELSE  y UPON NOT cond   FI ;
```

2.3.3.6 The $\boxed{\text{WVR}}$ Operator

The $\boxed{\text{WVR}}$ operator (*"whenever"*) helps when we try to build a node which consumes datons at a ***faster*** pace than it produces them. – Using the UDF $\boxed{\text{IndexT}}$ from above, the exact definition of $\boxed{\text{WVR}}$ is:

$$(a \text{ WVR } k)_i = a_{\text{IndexT}(k,i)} \quad \forall\, i \in \omega$$

$\boxed{\text{WVR}}$ consumes both its operands synchronously. It scans its rhs. operand **k** until a $\boxed{\text{TRUE}}$ daton is found, and it picks then the daton of **a** with the same index. The latter daton forms the result daton of $\boxed{\text{WVR}}$. To obtain the next result daton, the scanning of the operands continues from the index where the previous evaluation left off.

The $\boxed{\text{WVR}}$ operator is semantically equivalent to the following UDF:

```
Wvr (a, k)  =  IF   FIRST  k   THEN  p  ELSE  q  FI
               WHERE p = a FBY q ;
                     q = Wvr (NEXT a, NEXT k) ;
               END ;
```

As a typical use of WVR, here is the UDF Clean which filters out any immediate repetitions of datons:

```
Clean (a)  =  a WVR  (TRUE FBY (a NE NEXT a)) ;
```

2.3.3.7 The ASA Operator

The ASA operator ("*as soon as*") is semantically equivalent to the following UDF:

```
Asa (a, k)  =  FIRST (a WVR k)  ;
```

ASA consumes both its operands synchronously. It scans its rhs. operand **k** for the earliest TRUE daton, and it picks then the daton of **a** with the same index. The result of ASA is a constant history generated from the latter daton.

The exact definition of ASA is obtained by applying FIRST to WVR:

$$(a \; \mathrm{ASA} \; k)_i \;=\; a_{\mathrm{indexT}(k,0)} \qquad \forall\; i \in \omega$$

2.4 The Semantics

So far, this chapter has taught us how to write meaningful Lucid programs. Thanks to the analogy of the plastic balls, we can even imagine how our programs might be executed. We must be careful not to overrate this analogy; it is by no means the authoritative definition of the Lucid semantics. The analogy extends to a further point, still: any of our plastic balls can be empty, in which case it ***provides no information***. (The reason why the information is missing is another matter.) Such "no information" datons are called **bottom**, the symbol is $\perp$. Correspondingly, a history can have $\perp$ components. A bottom daton carries less information than a proper daton; we say it is **less defined**. Based on this ***less defined*** ordering, a partial order is defined among histories (the history consisting only of bottoms takes obviously the lowest place).

Lucid can be understood as a single–assignment language: one history is assigned to each variable, once and for ever. The Lucid semantics is defined as follows:

> **The result of a Lucid program is the *least fixed point* history satisfying all the definitions in the program [AsW79a, AsW80]. (Least fixed point means here: the minimum history with regard to the partial order.)**

It is common to define variables recursively:

```
q  =  Func (q) ;
```

There may be a history **q**, so that $\boxed{\text{Func (q)}}$ is ***more defined*** than **q** (with regard to the partial order). This history **q** is unique. If such a history does not exist, **q** is $\perp$ throughout. – For example:

```
h = 1 FBY h+1 ;
```

h_0 is evidently defined; whenever **h** is defined up to an index i, it is also defined up to the index $i+1$ $\forall$ $i \in \omega$. By induction, **h** is therefore defined everywhere.

This variable **h** is actually an example for a special case where a particularly convenient translation (viz. ***pipeline***) is possible: no daton value of **h** is defined in terms of its own ***successor*** datons.

2.5 Program Execution

The term **Data Flow** designates the description of computations through datons moving through a net; we abbreviate Data Flow into **DF**. Histories are infinite objects, though no computer is able to operate directly on infinite objects. We have to re–organise the computations so that we need to operate only on individual datons, one after another. Let us now study the two strategies in which a DF program can be executed.

Data Driven DF

The strategy described in our analogy is called **data driven** DF (most researchers mean specifically ***data driven*** DF when they say "Data Flow"). The image of datons streaming down the arcs is particularly appropriate for data driven DF. CONSTANT and READ nodes are the original sources of datons, and they are eagerly feeding datons into the net. As soon as the required operand datons are available for a node, it is free to compute and produce its result. Such data driven nodes can, in general, not influence the arrival rate of their operand datons. The WRITE node has no dominance over other nodes, but simply writes out the datons which happen to arrive. A node may discard operand values (↑ end of 2.3.3.1), their evaluation was pointless, in retrospect. Data driven DF is inherently ***wasteful*** in this sense.

Demand Driven DF and Lazy Evaluation

Demand driven DF is a refinement of data driven DF, designed to be less wasteful than the latter. Further to the datons, demand driven DF has particles called **sitons** ($\zeta\eta\tau\omega$ = I request). Sitons travel *upstream* along the arcs, and each of them expresses the request for one daton. The WRITE node is the ultimate origin of all sitons; WRITE alternately issues a siton and receives a result daton. A CONSTANT or READ node produces a daton only upon receipt of a siton. All other node retain their daton handling capacity; however, they can now receive sitons at their outports and emit sitons from their inports, if appropriate. Sitons contain information about the nature of the request ("give me a daton with/without value"), and the nodes react accordingly. Unnecessary daton evaluation can be avoided in nearly all cases (↑ 5.6).

Once an evaluation has been instigated, by a siton, it may turn out that the daton value is not needed after all. In this case, a **lethon** is sent upstream to counteract the siton. Lethons (Lat. ***lethum*** = death) are close relatives of sitons; a lethon can be issued right after a siton, but before receipt of the response daton.

The nodes propagate lethons like sitons.

Demand driven nodes have considerable control over the producers of their operand datons; the $\boxed{\text{WRITE}}$ node has absolute dominance over all other nodes.

At present, most computers are von Neumann machines. Data Flow computations of either type can only be emulated on such a machine. A demand driven evaluator can be matched very closely to von Neumann machines, and it is possible to formulate this evaluator in quite acceptable von Neumann code. This is indeed what this thesis aims to achieve. – A form of demand driven evaluation has been used on von Neumann machines for a long time. It is widely known as **lazy evaluation** [HeM76], and it was first employed in LISP systems.

Even Data Flow machines do not contain moving streams of particles. They use in reality also an emulation, implemented in tailor made hardware instead of software. It is not very difficult to emulate demand drive on a data driven Data Flow machine [Sar82].

2.6 Deadlock

Every Lucid program produces an endless stream of datons, and nothing but a lack of input datons should be able to halt it. However, Lucid programs can contain faults which make them stop yielding results, permanently. Deadlock and livelock are such errors.

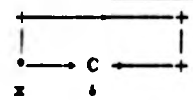
Deadlock is a type of programming error which re–emerges in almost all forms of programming. State σ is a **deadlock** state if:

– state σ can be left only if condition τ is TRUE, and

– condition τ is FALSE during state σ.

Section 2.4 stated which recursive definitions are constructive. Here is a pathological program, and its graph:

```
x  WHERE
     x = x ;
   END
```



One could say, the program defines x to be "whatever it happens to be". Consequently x is bottom throughout, due to the fixed point semantics. When this program is executed, an attempt is made to obtain the value of a daton x_i. Because of the cycle, a daton x_i can be evaluated only if x_i is known ***beforehand***; this is a deadlock (see also Cycle Sum Test, ↑ 6.1).

Another programming error is the livelock; **livelocks** are those computations which never deliver a result. In the following pathological example, $\boxed{\text{odd}}$ contains only odd numbers, and the UDF $\boxed{\text{Even}}$ is a filter for even numbers. $\boxed{\text{Even}}$ applied to $\boxed{\text{odd}}$ can never yield a result. Consequently, the result is bottom throughout:

```
Even (odd)  WHERE
              Even (x) =  x WVR ((x MOD 2) = 0) ;
              odd      =  1 FBY (2 + odd)
            END
```

CHAPTER III: Imperative Programs and Message Passing

3.0 Introduction

Whenever a program is executed on a digital computer, this is done in the form of numerous elementary **operations** (= computation steps, actions). The executing computer is characterised by the method in which the operations are set in motion, and each of these methods represents a computer **architecture**.

Historical Review (sketched)

John von Neumann developed the original stored program computer architecture (Moore school, EDVAC, 1945). But people tried immediately to make their machine even more *productive*, for example by allowing I/O transfers while the machine was busy computing the next result. This was achieved through ingenious technical fixes, which in turn provided a base for the invention of (pseudo–) concurrent computation. A computer system computes **concurrent**ly when it is simultaneously handling more than one computation. Later, after the dramatic growth in the number of computers, techniques were developed to *link* computers together. In this thesis we will give only little thought to the difference between real and pseudo concurrency.

Changes in hardware motivated the development of software, i.e. hardware took an active role, software a passive role. *Multiprocessing* operating systems were a reaction to the introduction of concurrent computation. Even today, designers of computer systems rarely pass the benefits of concurrent computation on to the applications programmer. The area has the reputation of being for experts only. This is in essence not justified, in fact the reputation stems largely from the use of unwieldy programming languages.

Nevertheless, programming techniques and languages for cooperating concurrent computations have been developed, mostly by academics. Every language reflects the priorities its inventor gave to the various aspects of concurrency. The various concurrent programming methods are best compared by discriminating between (A) how they set up concurrency and (B) how their concurrent units communicate. Early on, people were satisfied to have any provision for concurrency at all. Leaving genuine concurrency aside, we would place the UNIX® $\boxed{\text{fork}}$ primitive under (A) in this era in history. Similarly under (B), one would place in this era *shared* use of global variables. There are methods which are more refined. *Message passing* is the natural choice of communication method for concurrent systems with separate memories.

In **message passing**, the computing agents communicate solely by sending and receiving ***messages***, each **message** being a sequence of data. (We call each ***computing agent*** an "**actor**"; an actor is almost the same as a von Neumann machine. Full detail in 3.2.1.) The inherent modularity of message passing makes it attractive for quite general application.

Other concurrent programming concepts cater for aspects which are relevant in special situations. Making the data ***machine independent***, for example, is of great importance in inhomogeneous computer networks (Data Abstraction, CLU [Lis74]). Other researchers have at the same time tried to design languages which are much more amenable to ***analytic*** methods, and thus make program proofing a realistic idea. Most of these languages are built on very concise sets of fundamental constructs. Hoare's CSP [Hoa78], Brinch Hansen's EDISON and, in a different sense, Lucid belong to this category.

Criteria for the Implementation Language

On present–day computers, which language would be a suitable vehicle for implementing Lucid? Here are a few simple guidelines to aid us in our search for an appropriate language:

- Is the language comprehensive enough for the task in hand?
- Are the resulting programs easy to *read*? This thesis is meant to convince the reader that the translation is meaningful and correct, and a well readable language would support this aim. The "production" implementation language, on the other hand, may be arbitrarily cryptic.
- Is the language available on many computer systems? If not, would it be easy to implement, possibly by modification of an existing system? Programs written in a good popular language are easiest to understand and translate.
- Last, and least: are the language features a reasonable reflection of the way in which present–day computers work? Optimisation becomes unnecessarily difficult if this aspect is ignored.

Clearly, many candidates pass these simple guidelines equally well. We will see that Message Passing Actors (**MPA**, ↑3.2.1) support *modular* program design. The author had advance experience with MPA, and there was therefore a certain sympathy for MPA languages. There is little doubt that valid arguments can be brought in favour of other programming styles with cooperative concurrency. Various programming languages have been looked at and a decision for MPA has finally been taken.

We chose to design directly the language most convenient for our purpose. This language is called **LUX**. LUX has been developed to suit the translation algorithm. Various versions of LUX, each with its matching translation algorithm, have been tried out. We present here only the design which eventually seemed best.

Structure of this Chapter

In this chapter we look first at the von Neumann machine, the archetypal imperative computer. After that we introduce the crucial elements of any MPA language, namely actor *creation* and the primitives $\boxed{\text{SEND}}$ and $\boxed{\text{RECEIVE}}$. We look at variations of, and alternatives to, message passing actors. We look then at CSP as an instance of a MPA language, and we discuss its properties. (A variant of CSP has, for a while, been the candidate as target language. We show why it was found unsuitable.) We present finally the language LUX in full.

3.1 The von Neumann Machine

Most computers these days (1983) have essentially a von Neumann architecture. **Von Neumann machines** are *sequential* computers. There, only *one* operation can usually be active at any single moment. Although every pure von Neumann machine is sequential (non–concurrent) by nature, a certain degree of cooperating concurrency can be achieved, simulated or genuine, but only at rather high cost. We discuss von Neumann machines here only as far as relevant for implementing Lucid.

3.1.1 Flow of Control in von Neumann Architecture

The program (*code*) for a von Neumann machine is a directed graph, with **instructions** as nodes. Programs for von Neumann machines are called **sequential** or **imperative** programs. A classic von Neumann machine executes non–imperative programs either inefficiently or indirectly, through compilation. Lucid is a non–imperative programming language.

The flow of control formalism models the execution of a sequential program. The formalism assumes that per actor there is one token of computing activity. (An **actor** is something rather like a sequential program, * 3.2.1.) The token is usually called the **PC**, for *"program control"*. The PC moves along the arcs in the arrowed direction, with a defined starting point. Every instruction type is the encoding of an operator; the respective operation is performed when the PC reaches the instruction (= node). In other words: sequential programs state explicitly the sequence in which the operations are carried out.

The classic von Neumann machine has only one PC, and it can therefore only perform a single succession of operations. This can be expanded into ***concurrent*** computations by putting von Neumann machines side-by-side. The same effect can be approximated by switching one von Neumann machine between a number of actors; this is ***pseudo-concurrency***. Finally, cooperating concurrent computations are obtained by adding a means of ***communication*** to concurrent computations.

3.1.2 Handling of Datons in von Neumann Architecture

In von Neumann machines all the memory takes effectively the form of **storage cells** (traditionally and misleadingly said to be ***variables***). The contents of some storage cells change in the course of instruction execution.

The concept of ***histories*** is not all that alien to von Neumann machines. The values, successively held in a storage cell, can indeed be viewed as components of a ***history***. One could, for example, associate a "write" counter to each storage cell, and increment it whenever a new value is written into the cell; the counter would obviously tell the "daton index" of the currently stored value. This comparison presupposes that all Lucid nodes evaluate their histories in the order of increasing index (*"monotonically"*). Such nodes are, indeed, particularly easy to implement, viz. using pipelines. Some nodes, however, can leave the order of daton evaluation unspecified, namely when each of their evaluations is independent from all previous evaluations. The order of daton evaluation needs careful supervision only in nodes with memory, nodes which are not ***primitive***.

3.2 Message Passing Actors

3.2.0 Introduction

As stated before, message passing is the natural choice of communication method among separate computers. Hewitt et al [HBS77] proposed its use in a much more comprehensive context. Message passing enforces a high degree of modularity, and this is one of its strongest attractions. The term "*actor*" is due to Hewitt; actors will be explained in 3.2.1. There is great divergence of terminology in this field. Common terms in place of *actor* are virtual processor, process, task, and job. **MPA** is short for Message Passing Actors.

C.A.R. Hoare presented his Communicating Sequential Processes (**CSP**) in his report [Hoa78]. Combining pre–existing techniques in a new and rather elegant style is the main achievement of CSP. CSP is a semi–formal language, and message passing is one of its central primitives (↑ 3.3).

The Experimental Programming Language **EPL** [MaT79] was devised and implemented by the Warwick Distributed Computing Project Group. EPL was developed at roughly the same time as CSP, and it owes CSP more than Hewitt's actors. EPL is a *bare bones* language in the spirit of BCPL. It has been implemented on two different machines, and it was meant for experimenting with message passing. A typical EPL program would contain substantial lengths of code where only *conventional* computations are carried out without messages being passed.

The language **OCCAM** [Inm82] might be a candidate as the true implementation language; OCCAM is a descendant of CSP and EPL. The inventors of OCCAM see it as a new breed of assembler language, particularly suited for multiprocessor systems. The OCCAM actor creation and message passing are both *static*, which makes them too inflexible for what our translation requires. Lucid programs *without recursive UDFs* could be translated into OCCAM without too much difficulty. Appendix D shows an example of what would come out if our translation algorithm generated OCCAM code (unoptimised).

In this thesis we will extensively use a *purpose built* language named **LUX**. The MPA side of LUX has been strongly inspired by EPL. LUX will even be used as the yardstick in all our explanations and comparisons. This is intended only to avoid a flood of insubstantial definitions, and it must not be understood as a denigration of other languages. LUX itself is hardly free from imperfections, but it is very suitable for the task in hand. In the following all MPA examples will present the LUX case, unless otherwise stated.

Why do we invent yet another language instead of using an existing one? The language LUX has been designed for the sole purpose of legibly formulating the Lucid node acts. There are many other languages in which this could have been done. However, the truly popular languages contain generally no primitives for the kind of concurrency we need. (LUX "*exceptions*" resemble the *interrupts* of assembler languages, and "*doors*" are the LUX device for exception handling. Ordinary languages comprise no obvious elegant equivalent for LUX doors.)

The very popular language PASCAL [Wir71] forms the syntactic backbone of LUX. LUX has been obtained simply by enriching PASCAL with a number of extra features. There are two simple extensions right at the start:

- the underline character "_" is allowed in identifiers (it can make identifiers more readable),
- the special symbol [ACT] occurs in some places where in ordinary PASCAL one would write [PROCEDURE].

Here is a simple but complete LUX program. The program emulates the children's game with a triangular inequality: a stone (0) defeats scissors, it makes them blunt, paper (1) defeats the stone, it wraps it up, and scissors (2) defeat paper, they cut it.

```
ACT Act_Root_ ;
  VAR  a, b : ACTOR ;  ra, rb, win : INTEGER ;
  BEGIN
    a := CREATE (Act_Player_) ;
    b := CREATE (Act_Player_) ;
    REPEAT
      ( , ra) := RECEIVE FROM (a)     ;
      ( , rb) := RECEIVE FROM (b)     ;
      win     := (3 + ra - rb) MOD 3 ;
      IF   win > 0
      THEN writeln ('Point for player', win) ;
    UNTIL FALSE ;
  END ;


ACT Act_Player_ ;
  BEGIN   REPEAT
            SEND Choice012 TO (Creator) ;
          UNTIL FALSE ;
  END ;
```

(Both players' choices are taken and compared. Each player is free to base his choice on a long term analysis of the other player's behaviour. In the program, this decision taking is hidden in the parameterless function Choice012, which returns 0, 1 or 2.)

3.2.1 Acts, and Actor Creation

Acts, actors, and the creation and initialisation of actors will be introduced in this section.

Analogy (Food for Thought)

Every act is somewhat like a cooking recipe. Actor creation and initialisation corresponds to the preparations for cooking a meal (buying the ingredients), program execution is the cooking itself, and the program output is the meal. The actor is the combination of the ***ingredients***, in their current state of processing, and of a ***bookmark*** pointing to the line in the recipe to which the cooking has progressed. Many meals can be cooked from the same recipe, even simultaneously. These meals will be of separate identity but of equivalent nature.

Acts vs Actors

Every program with Message Passing Actors is written as a collection of **acts**, every act being a piece of sequential code. Every act definition has exactly the ***syntax*** of a PASCAL procedure declaration, only with the keyword PROCEDURE replaced by ACT. Acts are the largest building blocks of such a program. Here is a typical act definition:

```
ACT Act_xyz ;
    BEGIN
     ...        (* The body of the act. *)
    END ;
```

An **actor** is the sole framework in which computing action can take place, where **computing action** is meant to cover all CPU action in general. Actors are ***activations*** (= ***instanciations***) of acts. Let me repeat that acts are mere descriptions of computing action. Many people have great difficulty in distinguishing between acts and actors, though they are in essence different kinds of objects. Executing an act would be as pointless as boiling a recipe, in our analogy. If you are hungry, it is not enough to buy a cookery book (set of acts), you need the ingredients as well. Only the synthesis of the two (the actor) can eventually give you a meal (computation result).

Every act is a global constant in LUX. The identifier of an act must only occur in CREATE instructions, but never in assignments or messages. Actor names, on the other hand, are not constants but are data values of type ACTOR; there are no extra restrictions to their use. Our translation requires no ***nested*** act definitions.

Actor Creation

A LUX program, a set of acts, is like the definition of a set of mathematical functions. A definition on its own can not yield a result. A mathematical function yields its result only when applied to a sequence of operands. The **actor creation** is the corresponding operation which sets computing action in motion. Every actor is generated by applying the $\boxed{\text{CREATE}}$ operation to an act. Each actor has its individual actor **name**, which is something rather like an address. If $\boxed{\text{Act\_xyz}}$ is an act, and if $\boxed{\text{pqr\_actor}}$ is a storage cell which can hold the name of an actor, then

```
pqr_actor := CREATE (Act_xyz, h0, h1) ;
```

creates a new actor from $\boxed{\text{Act\_xyz}}$, and stores the name of this new actor in the storage cell $\boxed{\text{pqr\_actor}}$. Actually, an actor can carry out its computations even if its name is not known to any actor. However, the name of an actor is needed when it communicates with other actors (↑ 3.2.2). Numerous actors can stem from (can be created from) the same act.

The act specifies the operations which are carried out by the actor, with execution starting at the beginning of the act. Every actor starts ***acting*** (i.e. computing) at the moment of its creation. An actor ***terminates*** forever once execution reaches the end of the act (where PASCAL procedures would instead do a "call return").

In the $\boxed{\text{CREATE}}$ instruction, further actual operands may be appended after the act specification ($\boxed{\text{h0}}$ and $\boxed{\text{h1}}$ in our example above). These extra operands are passed to the actor like procedure parameters. They re-emerge, completely untouched, as values for the formal operands (example: ↑ 3.4.4). In our translation, these extra operands are always constants. Names of communication partners (operand actors) are never passed in this way, but only via the actor initialisation (↑ 4.1).

Actors have no particular representation within the LUX syntax, since they are not syntactic objects; they can only be characterised by the operations applicable to them. The only possible operations on an actor are: its own creation, sending a message to it, receiving a message from it, and assigning its name to a storage cell.

Each actor is characterised by the pair <*act, memory*>. An actor can share its act with other actors, but every actor has its dedicated memory (i.e. actors can be "brothers").

Actor Head

All actors run under a ***runtime system*** which takes care of actor creation, scheduling, message passing and further administration. In the course of actor creation, the supervisor allocates a record (i.e. some storage space) called the **actor head**. The actor head holds information about the particular actor. The contents of the actor head changes during execution.

We are only interested in very few items within the actor head, and it is sufficient to assume that actor heads are pre–declared as follows:

```
TYPE

   MSGTYPE        = (DATON, READY, COMPUTE, NULLIFY, ADVANCE) ;

   ACTOR          = ^ ACTOR_HEAD ;     (* a possible def'n of ACTOR *)
   ACTOR_HEAD     = record
      creator     : ACTOR    ,
      xrequest    : MSGTYPE ;              (* prestored with READY  *)
      xindex      : INTEGER ;
      (* There are various further pieces of information           *)
      (* which are used for administration                         *)
      (* (but which are inaccessible to the user)                  *)
      (* scheduling status, intrinsic priority, actual priority,   *)
      (* program counter, stack pointer (for procedure binding)    *)
    END ;
```

Some ***special functions*** are provided through which each actor can obtain useful information about itself. These functions are all ***parameterless***, and their result is ***actor specific***. For example, the function $\boxed{\textbf{Myself}}$ yields the actor name of ***this*** actor itself, $\boxed{\textbf{Creator}}$ yields the name of the actor which ***created this actor***, and $\boxed{\textbf{Reveal}}$ is a multivalued function yielding the entire message of the last exception, i.e. $\boxed{\textbf{xrequest, xindex}}$. If used as a single valued function, $\boxed{\textbf{Reveal}}$ yields just the contents of $\boxed{\textbf{xrequest}}$. Through these functions, the actor can get access to the information in the

actor head. Actors are neither capable to explicitly *change* any actor head nor to inspect heads of *other* actors.

Throughout this thesis, the *leftmost component* of every message (↑ 3.2.2) is of type $\boxed{\text{MSGTYPE}}$ and indicates the nature of the message. The message is a request if that component is $\boxed{\text{COMPUTE}}$, $\boxed{\text{NULLIFY}}$ or $\boxed{\text{ADVANCE}}$; it is an exception request if that component is $\boxed{\text{NULLIFY}}$ or $\boxed{\text{ADVANCE}}$. *Daton values* are passed around by messages whose first component is $\boxed{\text{DATON}}$. The message type could be indicated in other ways than via the first component, but *this* method has the advantage that every message is easy to identify (↑ 5.3.2).

$\boxed{\text{READY}}$ does not occur in *messages*, but the cell $\boxed{\text{xrequest}}$ in the actor head can be set to $\boxed{\text{READY}}$, thus indicating a particular actor status ($\boxed{\text{xrequest}}$ is initially set to $\boxed{\text{READY}}$).

As we said above, the act may have formal operands, and they are prestored with the *extra operands* from the $\boxed{\text{CREATE}}$ instruction.

Root Actor

We are now in a chicken–and–egg situation. $\boxed{\text{CREATE}}$ is an *operator*, and operators occur only in *acts*. However, the execution of any operator (such as $\boxed{\text{CREATE}}$) must be *preceded* by the creation of the actor in whose act it occurs. This problem is easily solved, the LUX program execution is set running by the *implicit* execution of:

```
root_actor := CREATE (Act_Root_) ;
```

The LUX program must therefore contain a definition of the $\boxed{\text{Act\_Root\_}}$. The $\boxed{\text{root\_actor}}$ creates further actors, *all* computing action has its ultimate origin in this actor. Incidentally, the storage cell $\boxed{\text{root\_actor}}$ is not accessible from anywhere, there was simply no need to make it accessible. Unlike PASCAL programs, there is no *main program* section in LUX programs; $\boxed{\text{Act\_Root\_}}$ takes this role instead.

Miscellany Concerning Actor Creation

So far we have described **dynamic** actor creation, i.e. actor creation through a run time operation. The alternative is **static** actor creation, where actors are pre–created before computing action has started anywhere in the program. Static actor creation can be ***simulated*** by dynamic actor creation, whereas the inverse is not possible. In this thesis we need dynamic actor creation for the implementation of ***recursive*** Lucid UDFs.

Actor **initialisation** is usually the first thing to follow right after an actor has been created. In the initialisation, the new actor is provided (through messages, mostly from its creator) with various information which it needs to go about its job. Among this information will normally be the names of the ***communication partners***. Some actors (e.g. root_actor) contain nothing which needs to be initialised.

3.2.2 SEND and RECEIVE

The LUX inter–actor communication method is **unbuffered message passing** between pairs of actors. A **message** is any sequence of data items. **Unbuffered** means that the message is passed if one actor wants to SEND and if ***at the same time*** the other actor wants to RECEIVE. Furthermore, if the SEND or RECEIVE instruction names a particular message sender or receiver, the actors involved must match what is asked for. If an actor comes to a SEND or RECEIVE instruction, it ***waits*** until all the preconditions for communication (just mentioned) are satisfied. Once the message has then been transferred, the sender and the receiver can both resume execution.

The instructions SEND and RECEIVE are the message passing primitives. They "dictate" to the system that a message shall be sent or received. At any single moment an actor can either be ***computing***, waiting to SEND, or waiting to RECEIVE. The primitives have in general the following form:

- the $\boxed{\text{SEND}}$ instruction states what the message is, and to which actor(s) the message shall be sent,
- the $\boxed{\text{RECEIVE}}$ instruction states which actors are eligible as message senders, and where the message shall be stored.

For example, LUX has three message passing instructions:

```
SEND e0, e1, ... en TO ( receiver0, receiver1, receiver2 ) ;
```

This instruction specifies a transfer of the message ($\boxed{e_0}$, $\boxed{e_1}$, ... $\boxed{e_n}$, where each $\boxed{e_i}$ is an expression) to a set of receiving actors; brackets may enclose the message. Any number (minimum is one) of receivers is permitted; in our example there are 3 of them. The receivers must ***exist*** while the $\boxed{\text{SEND}}$ is in execution. The execution of the $\boxed{\text{SEND}}$ instruction is complete when the message has been accepted by ***each*** of the quoted receiving actors. The quoted receiving actors must all be different.

```
(sender, c0, c1, c2, ... cn) := RECEIVE () ;
```

This is the instruction for an ***undirected*** $\boxed{\text{RECEIVE}}$. It is best understood as a ***multiple assignment***, like from a multivalued function. (The storage cells on the left of := must have been declared elsewhere.) It means: as soon as a message arrives, from any actor, it is stored in the $n+1$ storage cells $\boxed{c_0}$, $\boxed{c_1}$, ... $\boxed{c_n}$ (word by word, progressing from $\boxed{c_0}$ to $\boxed{c_n}$; how many values are stored is determined by the ***left*** hand side). If the receiving instruction asks for ***fewer*** message components than provided in the $\boxed{\text{SEND}}$ instruction, the remaining components of the message will be ***lost***. If the receiving instruction asks for ***more*** message components than provided in the $\boxed{\text{SEND}}$ instruction, the remaining storage cells on the receiving side will be filled with ***unpredictable material***. – The sending actor's name is stored in the leftmost storage cell (here: $\boxed{\text{sender}}$), i.e. it is "stuck in front" of the message. If more than one sender is simultaneously ready to send, one sender is chosen ***at random***, and all other senders continue waiting until successful at some later time. Any message component can be ignored by leaving its field ***empty*** in the assignment (but not omitting the ***comma***), as in:

`( , , component3) := RECEIVE () ;`

`(sender, c0, c1, ... cn) := RECEIVE FROM ( sender0, sender1 ) ;`

This is the ***directed*** RECEIVE instruction. The message can come only from any sender actor specified after the FROM. There can be any number (minimum: one) of sending actors. These senders must exist ***while the*** RECEIVE ***is in execution***. Everything else is exactly as in the undirected RECEIVE instruction.

In LUX, messages can consist of values of arbitrary type, and even actor names are allowed. Pointers, arrays, or names of procedures, function, or acts are ***not*** allowed as messages components. LUX **requests** are particular messages, they are of importance in translated Lucid (explanation: ↑ 4.2). Section 3.4.2 describes the LUX mechanism for passing "exception" messages.

Every act is a ***global*** constant in LUX. Every act is therefore permanently known to every actor, whereas it is not permitted to make an act known to another actor by transferring it in a message. In LUX, the use of global objects other than constants is generally frowned upon; actor names are clearly not constants.

The situation can arise where a number of actors try simultaneously to SEND to the same RECEIVE, i.e. all these senders fulfill equally the preconditions for a message transfer. It has been stated above that in such a situation one of the senders is chosen at random, and the remaining senders keep waiting for further RECEIVE. LUX does not specify any order (e.g. "first come first serve") because that would in general not be enforceable [Lam78].

3.2.3 Contentious Points with Message Passing

Message passing can give problems in ***typed*** languages, because the words in the different possible messages can be of ***non-uniform*** type; this problem did not exist in EPL since it is type-less. In LUX, we glance over this problem by assuming that the types of the message and of the left-hand side of the RECEIVE instruction ***do*** match. This can be ensured by run time checks.

Our translation process generates code in which type clash errors cannot occur. If one wished to change LUX into a general-purpose programming language, one could define: every RECEIVE instruction assigns an entire ***structure***, where the structure can be of ***union*** type.

Deadlock (↑ 2.6) is another problem area for message passing, and for concurrent programs in general. (Our translation algorithm generates deadlock-free code, as long as the Lucid program is flawless.)

3.2.4 Variations of Message Passing

Message passing, as presented so far, can obviously be varied in a number of ways. We study only substantial variations.

The addressing of senders and receivers is a rich field for variation. **Broadcasting** is of particular interest, i.e. the simultaneous sending to all receivers. (The SEND instruction of LUX allows sending to a ***set*** of actors.) If broadcasting is done in unbuffered message passing, its effect must be defined on receivers which are currently not waiting (will the sender wait for them all?). — There are also uses for a "lottery SEND", which sends to a set of receivers, but eventually gives the message to only one of them.

Non-determinacy can go further than merely leaving it open from which actor to receive a message. It has been said in 3.2.2: at any single moment an actor can either be purely computing, waiting to SEND, or waiting to RECEIVE. There are relevant applications which would benefit if ***more than one*** of these were simultaneously possible. A ***priority*** rule might be provided for the case where $\boxed{\text{SEND}}$ and $\boxed{\text{RECEIVE}}$ become simultaneously enabled. Obviously, pure computing must get the lowest priority since it is permanently enabled.

We could redefine the measures taken if one actor wants to send a message without the target actor being ready to receive it. Instead of letting the sender ***wait***, the receiving actor could **buffer** the message, and let the sender proceed immediately. To be general, the buffer should be ***unbounded***. – Obviously, this ***buffered message passing*** is much more complex to implement than the unbuffered variety, and its fundamental operations are less directly related to the "inborn" operations of conventional computers. The extra luxury in the buffering must usually be weighed against some ***extra cost***. Often enough this luxury is not even wanted. As an example for the latter, here is a piece of LUX code with a useful effect which would be much harder to achieve if message passing was buffered:

Example ($\boxed{\text{Act\_Guardian\_}}$): unbuffered message passing

```
ACT Act_Guardian_ ;
    VAR sender1, sender2 : ACTOR ;
    BEGIN
      REPEAT
         sender1 := RECEIVE () ;
                         (* No other message sender can now get in. *)
         sender2 := RECEIVE FROM (sender1) ;
      UNTIL FALSE ;
    END ;

VAR guardian_actor : ACTOR ;    (* must appear in the declarations *)
    guardian_actor := CREATE (Act_Guardian_) ;
```

This $\boxed{\text{guardian\_actor}}$ toggles between its ***two states*** every time it has received a message. Initially, it waits for a message from anywhere; the message could be produced by:

$\boxed{\text{SEND 0 TO (guardian\_actor) ;}}$

Once the initial message has been received, a second message is expected from the same sender (viz. from $\boxed{\text{sender1}}$). If other actors try to send to the $\boxed{\text{guardian\_actor}}$ while it is in this state, they are forced to ***wait*** at least until it has returned to the initial state. The $\boxed{\text{guardian\_actor}}$ returns to the initial state once the second message has been received. The message itself is ignored throughout, only the ***event*** of the message matters.

Semaphores

Actors created from $\boxed{\text{Act\_Guardian\_}}$ can ensure that a certain ***access right*** is given to only one actor at any single moment. For example, they can be used to prevent multiple ***simultaneous alteration*** of shared memory (disastrous!). If a number of actors want to eat biscuits from a common box of biscuits, this would be safe if each of them followed the pattern:

```
SEND 0 TO (guardian_actor) ;
IF   any biscuits left?
THEN eat one biscuit  ;
SEND 0 TO (guardian_actor) ;
```

The $\boxed{\text{guardian\_actor}}$ is an MPA style implementation of ***semaphores*** (↑ 3.2.5)

3.2.6 Concurrency Methods other than Message Passing Actors

Concurrent computations can communicate through means other than message passing. We ignore here concurrent computing on specialist computers (CRAY, vector processors) altogether.

We mentioned before that the most straight–forward and simple–minded communication method is the use of ***shared memory segments***. This method can be hazardous when used carelessly, for example when two actors change shared memory in a time overlap. This can be brought under control by the use of

SEND 0 TO (guardian_actor) ;

Once the initial message has been received, a second message is expected from the same sender (viz. from sender1). If other actors try to send to the guardian_actor while it is in this state, they are forced to ***wait*** at least until it has returned to the initial state. The guardian_actor returns to the initial state once the second message has been received. The message itself is ignored throughout, only the ***event*** of the message matters.

Semaphores

Actors created from Act_Guardian_ can ensure that a certain ***access right*** is given to only one actor at any single moment. For example, they can be used to prevent multiple ***simultaneous alteration*** of shared memory (disastrous!). If a number of actors want to eat biscuits from a common box of biscuits, this would be safe if each of them followed the pattern:

```
SEND 0 TO (guardian_actor) ;
IF   any biscuits left?
THEN eat one biscuit  ;
SEND 0 TO (guardian_actor) ;
```

The guardian_actor is an MPA style implementation of ***semaphores*** (↑ 3.2.5)

3.2.5 Concurrency Methods other than Message Passing Actors

Concurrent computations can communicate through means other than message passing. We ignore here concurrent computing on specialist computers (CRAY, vector processors) altogether.

We mentioned before that the most straight–forward and simple–minded communication method is the use of ***shared memory segments***. This method can be hazardous when used carelessly, for example when two actors change shared memory in a time overlap. This can be brought under control by the use of

semaphores or **capabilities** ([Fab68, Wil72]); there, any actor must hold (like a token) the ***exclusive access right*** to the shared memory while changing it (protected regions, MODULA [Wir75]). – Programs using shared memory may be very efficient (fast), but the method is not applicable in all distributed computer systems. Anyway, shared memory must never be used in other than a very disciplined manner [WuS73]. Some languages enforce such discipline through special constructs, such as the modules [Hoa74] in MODULA and the clusters in CLU [Lis74].

Coroutines are in effect a subset of message passing actors, though, historically speaking, coroutines are of independent origin. Terms like "coroutine technique/method/style" are often used in the rather general sense of "multi–actor technique/method/style".

A computer with one architecture can acquire the outer appearance of a computer with totally different architecture either through some form of ***translation*** or through an ***interpreter*** (program). – There is reason to assume that user–specific **microcodes** will be commonplace in the next computer generation. It will then be possible to choose the most suitable architecture for each computation, and to emulate that architecture through a tailor–made micro–coded interpreter. Once the Lucid machine, say, has been implemented well, one will no longer have to worry about optimal translation into imperative code. Through the microcode, the interpreters will also be able to make full use of advanced computer hardware, for example, of **associative memory**.

It can be shown that all communicating concurrent programming methods are essentially of equal power, i.e each method can be simulated within each other method.

3.3 Hoare's CSP

With his Communicating Sequential Processes (**CSP**, [Hoa78]), C.A.R. Hoare introduced a concise notation which made message passing more amenable to scientific study. The merit of CSP lies in the achievement of great computational power from a small set of primitives. (CSP has similarities with Hewitt's PLANNER–73.) Hoare's paper [Hoa78] deserves praise for openly anticipating practically all points of criticism of CSP. Hoare disclaims expressly that CSP is meant to be a "production" programming language.

CSP programs are based on fixed sets of actors. There is no recursion, nor are actor names allowed as data values. Each SEND or RECEIVE operation must explicitly quote exactly one communication partner (actor). The CSP message passing is unbuffered.

The difference between acts and actors is not very prominent in CSP. CSP has means to make sequences of instructions (i.e. acts) into actors or even arrays of actors. CSP uses a very concise notation, all operators are denoted by short symbols. Here are the most essential primitives (merely an approximation; CSP message passing refers in reality to channels, not to actors):

X?c

This is the ***receive*** instruction, **X** specifies the sender (actor), **c** is the storage cell in which the message will be placed. The receive instruction provides simultaneously a ***test*** (the ***input guard***) whether input is currently available. An ***undirected*** receive instruction is not provided.

Y!m

This is the ***send*** instruction, **Y** specifies the receiver (actor), **m** is the message (an expression).

$\boxed{* \; s}$

This expresses endless ***repetition*** of instruction **s**.

$\boxed{a_1 \;\|\; a_2}$

This expresses the creation and ***concurrent*** execution of two actors, where a_1 and a_2 are the respective acts (more precisely: a_1 and a_2 are the actual pieces of code).

$\boxed{[\, g_1 \rightarrow s_1 \;\; \square \;\; g_2 \rightarrow s_2 \,]}$

This is the ***alternative command***, the $\square$ separates the alternatives, g_1 and g_2 are **guards**, s_1 and s_2 are sequences of instructions. Each guard g_i is essentially a boolean expression [Dij75]. All the guards g_i in the alternative command are evaluated, and the s_i of all ***those*** alternatives are shortlisted whose guards evaluate to TRUE. One of the shortlisted alternatives is then chosen ***non-deterministically***, and it is executed.

Some instructions yield a ***truth value***, telling whether the instruction has been executed successfully or not. For this reason, it is possible and meaningful to place such an instruction as a guard.

The lack of certain facilities in CSP makes it virtually useless for the implementation of full Lucid. CSP has no ***dynamic*** actor creation, and this rules out the translation of recursive Lucid UDFs. Even the mere creation of numerous actors from the ***same*** act can only be done within a very rigid pattern. This would be an unjustified burden for our translation process.

Neither multiple SEND, nor undirected RECEIVE or multiple directed RECEIVE exist in CSP. They can, however, be laboriously constructed out of the given primitives. The lack of these facilities can thus be overcome, at a price.

Taken together, CSP is rather unsuitable as the target language for the translation of Lucid, since important facilities are not provided. Moreover, certain very common operations can be expressed only indirectly, by means of extra actors. The use of CSP might lead to illegible code.

3.4 The Language LUX

All the imperative code in this thesis is formulated in the language **LUX**. LUX is solely meant as the vehicle for expressing the result of our translation. Clearly, LUX must not be seen as a proposed new programming language, in competition with SIMULA 67 [DMN68], Concurrent PASCAL [BrH75], MODULA [Wir75], ADA etc. We allow therefore aesthetic imperfections in the language, as long as they bring advantages in other respects.

The provisions for ***non-determinacy*** in existing languages force the programmer into formulations which are often remote from the way in which computers work. For example, there is usually no proper counterpart for interrupts or exceptions (**exceptions** are CPU-internal interrupts, e.g. "***division by zero attempted***"). This will be put right in LUX.

The syntax of LUX is exactly that of PASCAL [Wir71], albeit with a few extensions. PASCAL has been chosen because of its current wide spread popularity. The reader's familiarity with PASCAL is taken for granted. The extensions aim to provide the type of concurrency which can be very easily transferred into reasonably efficient code on any present computer. The extensions have furthermore been designed to have the least damaging effect on program size and ***legibility***. It is rather obvious that the translation algorithm of this thesis will in most instances be implemented in languages ***other*** than LUX.

3.4.1 The Extensions of PASCAL

A few superficial extensions have been mentioned in 3.2.0, they are:

- the underline character "_" is allowed in identifiers,
- the special symbol ACT occurs in *some* places where in ordinary PASCAL one would write PROCEDURE (see also ↑ 3.4.3),
- Act_Root_ replaces the role of the PASCAL *main program* section,
- RETURN stands for a GOTO to the end of the act.

The substantial extensions can be grouped into the following topic areas:

concurrency: CREATE, acts, actors, initialisations,

cooperation: SEND, RECEIVE,

exceptions: EXCEPTION, doors, Reveal, RESET.

The first two have already been dealt with exhaustively. It remains only to explain the last point, exceptions.

3.4.2 The Exception Feature

What is Nullification?

In multi-process operating systems like UNIX®, the user can ***concurrently*** execute a number of programs, for example: edit one program ***while*** another program is being compiled. The user can also instruct the operating system to ***discontinue*** one of its current activities (the user might suddenly have found a reason why the whole compilation is pointless). Such a termination entails usually some form of ***clean–up*** phase, in which all perfunctory resources are released, for example: files are closed, memory is de–allocated.

As a variation of termination, one could think of a request which tells an actor to **nullify** an ongoing computation, i.e. to go back to a particular previous state. Some clean–up may be necessary for undoing modifications which have meanwhile been carried out, due to computing action. ***Irreversible*** state changes are carried out only right after the result acknowledgement; then, nullification is immaterial.

Situations similar to nullification appear in the LUX code for Lucid programs. For example, each instance of the OR operator requires the concurrent evaluation of the OR operands. As soon as the evaluation of either operand yields TRUE, the other operand is no longer needed, and its evaluation will be nullified. Again, the nullification can entail a clean–up phase, since inferior actors may have to be nullified, and memory must be put into a coherent state. In LUX, nullification (↑ 4.2) is the most important instance of an ***exception***. Nullification is clearly different from actor ***termination***: nullification merely puts the actor into a particular state (which is defined in the act) but does not eradicate the actor. (A further point regarding NULLIFY will be discussed in section 4.7.)

Technicalities

Exceptions make sense only with actors which stand for Lucid nodes (we will call such actors **node actors**). Every actor has in its actor head a cell `xrequest`, in which its exception state is recorded. The actor creation stores `READY` in this cell. "***The actor is in exception mode***" is synonymous with:

```
xrequest <> READY
```

In the act, however, the actor head can be inspected only via the `Reveal` function (↑ 3.2.1), and the same test would thus be written as:

```
IF   Reveal <> READY   THEN ...
```

In the following, the syntax and ***meaning*** of LUX exceptions will be explained, applications of exceptions will be mainly presented in the next chapter. Exceptions may be an important feature of LUX but, after all, actors run most of the time ***without*** getting exceptions. It is therefore even more important that the ordinary (not—nullified) program execution in LUX does not suffer from an ***over—emphasis*** on exceptions. A special notation and execution mechanism has therefore been developed which keeps both the program legible and allows perfectly efficient program execution, both in the nullified and in the non—nullified case.

Doors

A trapdoor in a fairy tale castle can be ***blocked*** or ***active***. If it is blocked, its presence is hardly noticeable when one walks over it, but if it is active the effect may be dramatic. There are quite similar **doors** in LUX, and they are used for the handling of exceptions. Here is an instruction with a door:

```
i := i + 1 ;                 :5
```

The `:5` in this example is the door. Remember that in LUX (as in PASCAL) all labels have the form of unsigned integers, and the ***number on the door*** (we call it the **door target**) refers to such a label. Every door operates like a conditional `GOTO` instruction. If xrequest is READY, the door is to be ignored: it has no effect. If xrequest is ***not*** READY while the PC passes over the door, the door has the effect of a GOTO (i.e. GOTO 5 in our example).

Matters are slightly different if the door is immediately followed (dynamically) by a slow instruction. A **slow instruction** is any instruction whose execution may take a long time, like RECEIVE, SEND, CREATE or a procedure call (↑ 3.4.3). An actor can spend a long time working on a slow instruction, and during this time xrequest can cease being READY, due to an exception. The actor will therefore check ***concurrently*** whether xrequest is no longer READY or whether the slow instruction has been completed, whichever occurs first. A GOTO ... is carried out if the ***exception*** occurs first, and the slow instruction is of such design that its effects are nullified. There is ***no effect*** if the slow instruction succeeds first.

Every door is thus a shorthand for:

```
REPEAT
    IF   Reveal <> READY
    THEN GOTO door_target ;       (* door_target is 5 in our example *)
UNTIL  the subsequent instruction has been executed completely ;
```

(In a proper implementation one would not use ***busy wait*** for such a wait-door.) All ***fast*** instructions (assign, add, multiply etc.) are permanently ready anyway, and the loop would in those cases be unnecessary.

It is sometimes required that a group of instructions be executed as an unbreakable ***entity***. This can be achieved simply by not placing doors inside the group.

The Implicit RECEIVE

We still have to define clearly how xrequest changes value. Above (↑ 3.2.1 "Actor Head") we have defined actor heads, and MSGTYPE:

```
TYPE
   MSGTYPE = (DATON, READY, COMPUTE, NULLIFY, ADVANCE) ;
   ACTOR_HEAD   = record
      creator    : ACTOR   ;
      xrequest   : MSGTYPE ;      (* prestored with READY  *)
      xindex     : INTEGER ;
      (* etc etc *)
    END ;
```

Only messages whose first component is NULLIFY or ADVANCE are exceptions, which is why we call them **exception requests**. Exception requests are issued by the instruction:

```
EXCEPTION e0, e1, ... en TO ( receiver0, receiver1 ) ;
```

which differs from the ordinary SEND instruction (↑ 3.2.2) only in the new keyword EXCEPTION. The messages from EXCEPTION instructions are not received by ordinary RECEIVE instructions in the receiving actor, but use a portion of the actor head as a one–message buffer. They can be retrieved from there via the Reveal function. In detail:

An actor can receive an exception only while its xrequest is READY. This rule ensures that no exception is accidentally lost. Every actor is readily equipped with special code for accepting and handling of exception messages (this code forms part of the LUX system "behind the scenes", not part of the act). For an actor **Y** this code goes as follows:

```
IF   ( xrequest = READY )  AND
     ( actor X wants to issue an EXCEPTION to actor Y )

     (* The actor Y "gets an exception":              *)
THEN ( , xrequest, xindex) := EXCEPTIONRECEIVE ()
     (* EXCEPTIONRECEIVE has the obvious meaning.      *)

ELSE ( put the exception sender X on a waiting queue,
       try again after actor Y has executed a RESET    ) ;
```

We have defined that the actor is in **exception mode** exactly iff:

```
xrequest <> READY
```

Since $\boxed{e_0}$, the first component of the exception message, is either $\boxed{\text{NULLIFY}}$ or $\boxed{\text{ADVANCE}}$, receipt of an exception will necessarily place the actor in exception mode. The actor is permitted to set its own $\boxed{\text{xrequest}}$ to $\boxed{\text{READY}}$ (i.e. it declares itself ready to accept a new exception request) only by executing the instruction:

```
RESET ;
```

3.4.3 Procedures

PASCAL–like procedures (function procedures as well as ordinary procedures) are allowed in LUX, too; they are not superseded by acts and actors. In the MPA framework, function procedures resemble actors which exist merely during the handling of every single request. Procedures have ***no memory***. However, the calling actor can take care of the memory, and "import" it into the procedure with each call.

LUX deals with procedures as if they were to be macro–expanded. In terms of message passing, the procedure underlies the control of the actor which called the procedure. The name of ***that*** actor is used for all message passing during procedure execution, and there is only one common exception mechanism per actor. When we say "***the procedure gets an exception***" we mean that its actor gets an exception during procedure execution. The procedure can access items of the actor head ($\boxed{\text{creator}}$, $\boxed{\text{xrequest}}$ and $\boxed{\text{xindex}}$) as usual via the special functions $\boxed{\text{Creator}}$ and $\boxed{\text{Reveal}}$ (↑ 3.2.1).

$\boxed{\text{:R}}$, a special kind of a door, is provided for procedures. When such a door is encountered while $\boxed{\text{xrequest}}$ is not $\boxed{\text{READY}}$, a return is made from the procedure, and execution proceeds in the calling program ***as if execution had got hung up in the procedure call***. Execution continues in this case at (the target of) the door which ***directly precedes*** the procedure call. Exception handling is ***inhibited*** during any procedure call which is not ***directly*** preceded by a door. During the execution of

such door–less calls, the system pretends [xrequest] was [READY]. Procedures must be designed so that they nullify all their effects before using a [:R] door.

Example for a function procedure with [:R] doors:

```
FUNCTION GetDaton (index : INTEGER ; operand : ACTOR) : ANYTYPE ;
   LABEL 1 ;
   BEGIN
               (* Possibly hang up in SEND (unlikely though): *) :R
   SEND (COMPUTE, index) TO (operand) ;

               (* Possibly hang up in RECEIVE:                *) :1
   ( , , GetDaton) := RECEIVE FROM (operand) ;

   RETURN ;    (* normal RETURN even if exception occurred.    *)

1: EXCEPTION (NULLIFY, index) TO (operand) ;                       :R
   END ;
```

This very useful procedure sends a particular message to the operand actor, and awaits then the arrival of a reply. If an exception occurs ***before*** the reply has been received, the exception request will be propagated to the operand actor, followed by an exception return from the procedure. There is no special procedure action if the exception occurs ***after*** the reply has been received (we might wish to ***preserve*** the daton value) This function procedure will play an important role later on

3.4.4 Example Act

Here is a typical example of an act:

```
ACT Act_Scale_ (scaling_factor : REAL) ;
    (* This node actor multiplies each operand    *)
    (* daton with the constant scaling factor.    *)

    LABEL 1 ;
    VAR
       operand, superior    : ACTOR   ;
       request              : MSGTYPE ;
       index                : INTEGER ;
       daton_value, result  : REAL    ;

    BEGIN
      ( , , operand) := RECEIVE FROM (Creator) ;
                 (* End of initialisation, beginning of action. *)

      REPEAT
        WHILE TRUE
        DO   BEGIN                                              :1
                 (* We say the actor is "dormant" while it    *)
                 (* is waiting here; it is "busy" otherwise.  *)
             (superior, request, index) := RECEIVE () ;

                 (* Possibly hang up in "GetDaton":           *) :1
             daton_value := GetDaton (index, operand) ;

             result := daton_value * scaling_factor ;

                 (* Possibly hang up in SEND:                 *) :1
             SEND (DATON, result) TO (superior) ;
           END ;

                 (* Code for the exception handling:          *)
1:      (request, index) := Reveal ;
        IF    request = ADVANCE
        THEN  EXCEPTION (request, index) TO (operand) ;
        RESET ;
      UNTIL FALSE ;
    END .
```

Assume furthermore, that Act_Root_ contains:

```
scale_actor := CREATE (Act_Scale_, 9.0) ;     (* 9.0 = scaling factor *)
SEND (DATON, another_actor) TO (scale_actor) ;     (* Initialisation *)
```

Let us first study the scale_actor in the absence of exceptions. In the initialisation, the scale_actor is provided with the name of another actor to which it will send messages later on. The sender of the initialisation and the first message component ($\boxed{\text{DATON}}$) are known anyway, and can therefore be ignored.

After the initialisation, the $\boxed{\text{scale\_actor}}$ gets hung up in an undirected $\boxed{\text{RECEIVE}}$ where it awaits a request, i.e. a message telling it what to do. When the $\boxed{\text{scale\_actor}}$ receives a $\boxed{\text{COMPUTE}}$ request (i.e. a message whose first component is $\boxed{\text{COMPUTE}}$), it will first call $\boxed{\text{GetDaton}}$. This will in effect ***propagate*** the message unchanged to the operand actor ($\boxed{\text{another\_actor}}$), and will then await the delivery of the operand daton value. Once the operand daton value has been delivered, the $\boxed{\text{scale\_actor}}$ computes the result daton value by multiplying the operand daton value with the scaling factor, and this result is then ***sent back*** to the actor which issued the $\boxed{\text{COMPUTE}}$ request in the first place. Once that has been completed, the $\boxed{\text{scale\_actor}}$ resumes awaiting ***another*** request.

Whenever an exception occurs, the $\boxed{\text{scale\_actor}}$ ***abandons*** what it is doing at that moment. This particular example contains nothing which needs to be cleared up. Instead the actor can directly proceed to ***propagating*** the unchanged exception request to the operand actor. There is never a ***reply*** to an exception message, which is why no $\boxed{\text{RECEIVE}}$ instruction follows after $\boxed{\text{EXCEPTION}}$. Since nothing else needs doing, the exception state is ended with a $\boxed{\text{RESET}}$ and the $\boxed{\text{scale\_actor}}$ loops back to await the next request. The $\boxed{\text{scale\_actor}}$ can accept an exception even in the dormant state, i.e. the exception may occur even while the actor is not "busy" with work. We can state in general: for efficiency, exceptions should always be propagated at the earliest possible moment.

Note that a door is placed before each $\boxed{\text{SEND}}$ or $\boxed{\text{RECEIVE}}$ (actually, it is at the end of the preceding line). Some groups of instructions have to be executed as an unbreakable entity, and a door is placed only after the ***last*** instruction of the group. Clearly every instruction has been furnished, where advisable, with an "escape route" (viz. a door) for the event of an exception.

Note that a door is placed before each $\boxed{\text{SEND}}$ or $\boxed{\text{RECEIVE}}$ (actually, it is at the end of the preceding line). Some groups of instructions have to be executed as an unbreakable entity, and a door is placed only after the *last* instruction of the group. Clearly every instruction has been furnished, where advisable, with an "escape route" (viz. a door) for the event of an exception.

3.5 Summary of Chapter III

The popular cooperative concurrent programming methods are mere extensions of sequential programming. The extension has been achieved by "bolting on extra features". *Hardware* aspects of inter–actor communication dominate these languages, and the programmer is forced to bear these aspects constantly in mind.

The worst deficiency of programs in these languages is their inherent *illegibility*. Good programs are generally written as sets of *modules*, where each module is dedicated to a *sub–problem*; most problems can be broken into sub–problems. *Actors* are the modules in the above languages. It is rarely possible to confine each sub–problem to exactly one actor. One is forced to disect sub–problems into more or less mysterious code fragments which are then strategically placed in numerous acts. Given a non–trivial program written in one of these languages, only an expert can recognise what the program computes, and it is extremely hard to locate intricate programming errors.

The *advantages* of imperative programming languages can be noticed when trying to implement such a language on a *conventional* computer. All their advantages stem from their greater affinity to the von Neumann architecture:

– the languages are easy to implement, and

– efficient execution is easily achieved.

It has already been said in the introduction that the choice of implementation language for Lucid is not very critical. Message Passing Actors have been chosen as the target of our translation because they are ***stylistically*** not worse than the other concurrent programming methods, they are ***modular*** in a beneficial way, it is easy to ***implement*** them well, and they have already been tried exhaustively in substantial programming tasks.

CHAPTER IV: The Translation

4.0 Introduction

This chapter deals with the translation of a *Graph Lucid* program into an equivalent structure in *MPA* style, which is the central issue of the thesis. "*Equivalent*" means that both structures represent the same input/output function. Our translation is carried out in two stages:

Stage one consist in designing for each individual node type an equivalent act, a **node act**. The example of a LUX act (↑ 3.4.4) was indeed such an act, and the reader is advised to use that example, for the time being, as the model of a node act. Ready made acts will be presented for most of the *fundamental* operators of Lucid (↑ 4.5.4 ff), and a comprehensive description will be given how to construct the act for the other operators (↑ 4.5.2, 4.5.3, but also 4.3). The full description of the node acts is very technical; this is why we shelve it for a while and present it rather late, in sections 4.5 f.

In ***stage two***, the **translation proper**, an arbitrary Graph Lucid program is re-formulated entirely in terms of the acts from stage one. Stage two is very straight forward. We explain this stage of the translation before stage one (↑ 4.3).

Graphs contain nodes, but nodes can themselves be graphs. These amazing nodes are the **UDF nodes**, of course; they break out of our two-stage classification. The construction rule for UDF acts can be obtained from the translation rule for programs, with only a little adaptation. This is why section 4.3 contributes to both stages one and two.

Node actors, **requests**, and the **protocol** are central in our further deliberations. Acts are just one way of, statically, encoding the computing action, which is a dynamic object. Acts rarely provide a good picture of the ***dynamics***, i.e. of the underlying execution strategy. The protocol is such a strategy with regard to the inter–actor communication. The functioning of actor nets is widely determined by the protocol. The chapter starts therefore with a conceptual description of node actors and requests (section 4.1), and this is followed (in section 4.2) by the protocol specification.

4.1 Node Actors, Protocols and Requests

Node Actors

When we speak of **node actors**, we mean actors which emulate Lucid nodes. Every node actor behaves like a demand driven *computing station*. Usually, node actors form part of a *net* of cooperating node actors.

Protocols

Let us assume that such a net of node actors is given (construction algorithm: ↑ 4.3). Each of the node actors can be viewed as an *autonomous computing station*. We are left with the task of making these autonomous units *cooperate*, with the ultimate aim of producing a result. This can be achieved with the aid of a protocol. A **protocol** is a pattern of message exchanges between actors, i.e. a governing macroscopic pattern. The protocol serves to *control* the flow of information and also the execution of computations. Section 4.2 specifies the protocol to be used throughout this thesis. The design of this protocol will be aimed at *demand driven* evaluation (↑ 2.5); this will be generally assumed without further mention. The use of a *universal* protocol, among all node actors, is an essential precondition for the modularity of our translation algorithm. Every node actor adheres to this protocol; therefore, node actors need to know *nothing* specific about their communication partners.

Requests and Requesting

A **request** is just a particular message, and requests can be of various **request types**. The request type is, by convention, indicated by the first message component (↑ 3.2.1, "*Actor Head*"). Requests serve in general for dictating to a node actor which action it shall carry out. A **reply** (a message in the reverse direction) is given only to *some* request.

As stated in 2.5, *demand driven* means that the *driving force* for computing action emanates from the program output, in our case from the WRITE actor. The WRITE actor sends a particular request to another node actor ●, hereby stimulating ● into some particular action; the action varies with the requests. In order to satisfy the request, ● *in turn* may need to request from further node actors

This pattern of one node actor requesting from another can reappear down to *any* depth. While a computation is in progress, some actors are dormant while others are busy with computing action. Consequently, a momentary *hierarchy* exists among the busy actors; this hierarchy is constantly changing in the course of daton evaluation. The hierarchy is throughout built up of pairs of actors, namely **superiors** which issue requests, and **inferiors** which accept requests and "do their best" that the requests be ultimately fulfilled. An inferior can simultaneously be in the role of superior in a *subordinate* request. Obviously, the WRITE actor takes the top rank in the hierarchy (we assume throughout that there is only one WRITE node). Constant, READ and COPY outports rank lowest. Multi–inport superiors can have more than one inferior, any COPY node actors can have more than one superior.

General Pattern of Node Acts

Most node acts have the following overall layout (this is a simplification):

```
ACT Act_Example ;
   VAR
        inport   : ARRAY [0..9]  OF ACTOR ;
        superior : ACTOR   ;          (* ↑ 4.1                      *)
        request  : MSGTYPE ;          (* ↑ 4.2                      *)
        index    : INTEGER ;          (* ↑ 4.2                      *)
        result   : ANYTYPE ;          (* ↑ 4.6.1                    *)

   BEGIN       (* Initialisation of this actor (↑ 4.3.1):           *)
       ( , , inport[0], inport[1], ...) := RECEIVE FROM (Creator) ;

       (* Due to its low intrinsic priority (↑ 4.7) the node     *)
       (* actor will wait here until the first request arrives.  *)

          (*********************************************)
          (*  The X-part must be inserted here.         *)
          (*  It is executed only once, at the beginning. *)
          (*********************************************)

     REPEAT
       WHILE TRUE DO
       BEGIN
          (*****************************************)
          (*  The node actor is dormant exactly while *)
          (*  it is hung in the following RECEIVE:    *)
          (*****************************************)

       (superior, request, index) := RECEIVE () ;

          (*********************************************)
          (*  The Y-part must be inserted here.          *)
          (*  It is executed once per request.           *)
          (*                                             *)
          (* contains at the end:                        *)
          (*  IF    request = COMPUTE                    *)
          (*  THEN SEND (DATON, result) TO (superior) ;  *)
          (*********************************************)
       END ;

1:        (*************************************)
          (*  The exception part is placed here. *)
          (*************************************)
       RESET ;
     UNTIL FALSE ;
   END ;
```

(Due to the nature of Lucid, this layout is almost identical to the one independently discovered by Finch [Fin81].) The eternal WHILE loop in this layout reflects the fact that all node actors operate like ***endlessly*** running computing stations. Certain preparing actions may have to be carried out before the loop is entered. Such instructions are placed in the **X–part** of the node act. The X–part contains the loop initialisation, but it can even contain, for example, request RECEIVE instructions. The WHILE loop starts with the acceptance of an order for new work (by receiving a request). This work is then carried out; the pertaining instructions are contained in the **Y–part**. The Y–part may include the eventual giving back of the result to the superior (the ***reply***). Some actors need to ***retain*** information from preceding loop passes, others do not. In the latter case it is common to say that the actor has **no memory** (intended meaning: it has no ***long term*** memory).

In the event of an exception, a jump is made to the **exception part**. After some appropriate measures have been taken, the exception state is cleared by RESET, and the eternal REPEAT loop takes us back to the **dormant** state.

Theoretically, there is little need for actor ***termination*** in an endlessly running program. Actors need to terminate only for efficiency reasons: termination sets storage free for reuse in other actors. Section 6.3 deals with actor termination (KILL request).

4.2 Protocol Specification

Motivation

Before we study the protocol, let us identify what shall be achieved by our protocol. In a rather primitive implementation of Lucid there would be merely one request:

> "Start evaluating one daton, and deliver the daton value to me." This request will ultimately be followed by that value being sent in the reverse direction. The next request will automatically relate to the ***next*** daton.

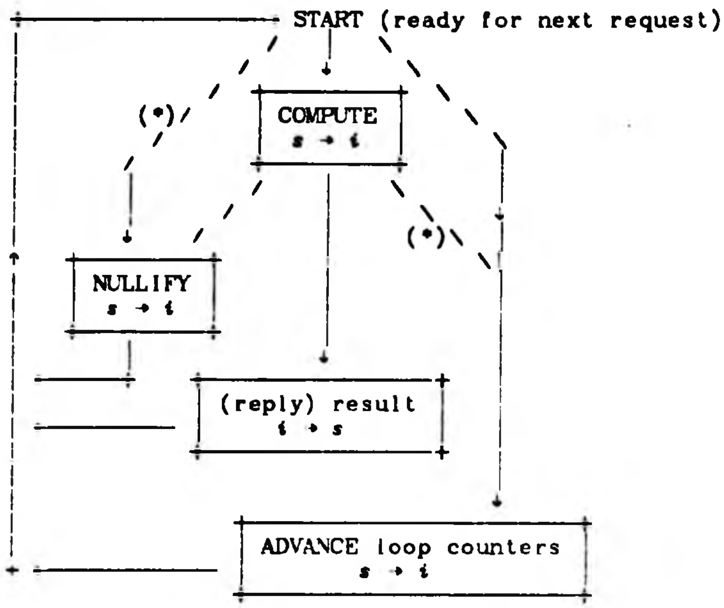
However, apart from being hopelessly inefficient, there are perfectly meaningful programs which would not be executable under this rudimentary protocol (e.g. any program with a concurrent OR in it, ↑ 1.1.3 and 4.5.3). We will not contemplate such a primitive implementation any further but aim for a protocol which is more refined in two respects. On top of the above request, we want to be able to do either of the following (Warning: don't take this as a definitive list of request types):

- ***Skip*** one daton. This is the same as asking for a daton without being interested in the actual daton ***value***. Such a request is essential for any serious implementation of the Lucid IF in pipeline DF.
- Once the computation of a daton value has been requested, one may suddenly want to ***nullify*** (annul, undo) that request for some good reason. Such a NULLIFY request is essential for the implementation of non–deterministic Lucid operators.

Furthermore, the protocol must take into account that any request can cause arbitrary ***subordinate*** requests. Higher–ranking evaluations can progress even while subordinate evaluations are ***under way***. Higher–ranking NULLIFY requests must be able to take proper effect on subordinate daton evaluations.

The Protocol as a Diagram

Let us now set out to answer the question: "in which sequence is the protocol executed, and where are variations possible?" Our range of requests is COMPUTE, NULLIFY and ADVANCE, and the following flowchart helps answering the question by showing the various possible ways in which the protocol can unfold:



(The paths marked (*) are never actually employed.) The symbol "*s* → *i*" in this diagram indicates that a message is passed from the *S*uperior to the *I*nferior. Execution starts at START, and the inferior is at this point assumed to be ***dormant***. It is furthermore assumed that both superior and inferior know constantly the index of the next daton to be computed. Both keep track of the current daton index, by a dedicated storage cell or similar means.

The flowchart makes no mention of the *action* itself. Each request has some action as consequence, e.g. evaluation of the daton value. This action starts with the reception of the request. START can be reached again once the action is complete.

The Protocol Requests

In detail, the requests are (all sent from a superior to an inferior):

$\boxed{\text{ADVANCE}}$

This request asks the inferior to advance the index counter by one (usually), namely to the successor daton. The previous daton will never again be asked for, it can be abandoned. – There is no reply to $\boxed{\text{ADVANCE}}$ requests.

$\boxed{\text{COMPUTE}}$

This request asks the inferior to ***evaluate*** the current daton (i.e. determine the value of the daton which is currently "due to come off the production line"). The inferior will take the measures necessary to obtain the daton value, at the end of which it offers to send this daton value to the superior. Under ***normal*** circumstances, the $\boxed{\text{COMPUTE}}$ is followed by the value delivery, and that is followed by an $\boxed{\text{ADVANCE}}$ request. There are, however, situations where the superior ***ignores*** the offer of the daton value and issues another overriding request (viz. $\boxed{\text{NULLIFY}}$). However, even after the daton value has been delivered there may be a renewed request for exactly the ***same*** daton. (This is why no automatic $\boxed{\text{ADVANCE}}$ request is incorporated in the $\boxed{\text{COMPUTE}}$ request.)

$\boxed{\text{NULLIFY}}$

This request asks the inferior to ***cancel*** any daton evaluation which may be currently going on in it (due to a $\boxed{\text{COMPUTE}}$ request). The state must be restored which existed before the evaluation of the current daton was requested. In our particle jargon, NULLIFY fires off a "kill token" ("***lethon***") which counteracts the preceding "***siton***" (↑ 2.5). – The $\boxed{\text{NULLIFY}}$ request is issued if the superior comes to a point where the daton value is no longer needed. Example: as soon as one operand of an $\boxed{\text{OR}}$ operation yields $\boxed{\text{TRUE}}$, evaluation of the other operand can be nullified. – There is no reply to $\boxed{\text{NULLIFY}}$ requests. We could even define $\boxed{\text{NULLIFY}}$

to have no effect on a dormant node actor, but instead we construct the acts such that $\boxed{\text{NULLIFY}}$ requests are never sent to a dormant actor.

Every request quotes, as its second message component, the **index** of the current daton. The initial index is 0, and the index must be ***changed*** only through $\boxed{\text{ADVANCE}}$ requests. It has been said, the index grows by ***one*** with every $\boxed{\text{ADVANCE}}$ exception. There is, however, the special index value $\boxed{\text{finalIndex}}$ which indicates that no further daton will ever be requested from the inferior. $\boxed{\text{finalIndex}}$ is a special constant, the infinitely large index ∞.

The index is at every moment equal to the number of preceding $\boxed{\text{ADVANCE}}$ requests, it would therefore be dispensable in the requests. Nevertheless, incorporating the index in each request offers a number of advantages:

- it can indicate the end of demand for a history, via $\boxed{\text{finalIndex}}$,
- nodes like $\boxed{\text{FBY}}$ can derive their state from the index, which relieves them from having memory,
- interfacing to ***tagged*** DF (↑ 6.5) becomes much easier,
- the index supports runtime checking and system error tracing.

If the inferior gets a $\boxed{\text{NULLIFY}}$ request while it is busy with $\boxed{\text{COMPUTE}}$ action (i.e. evaluation of a daton value) that action will be aborted. As specified in section 3.4.2, $\boxed{\text{ADVANCE}}$ and $\boxed{\text{NULLIFY}}$ requests are ***exceptions*** (unlike $\boxed{\text{COMPUTE}}$), and all evaluations are ***inhibited*** while any exception remains unresolved. From the superior's point of view, the action for $\boxed{\text{ADVANCE}}$ or $\boxed{\text{NULLIFY}}$ is indivisibly tied to the request, i.e. it would be pointless to ***delay*** the exception handling.

In a computation where successive daton values are needed, the normal cycle of operations is: $\boxed{\text{COMPUTE}}$ request, daton value delivery, $\boxed{\text{ADVANCE}}$ request. However, if a daton shall be consumed ***without*** its value being of relevance, the $\boxed{\text{ADVANCE}}$ request is issued directly ***without*** the preceding $\boxed{\text{COMPUTE}}$. We call such a request a **bare**

$\boxed{\text{ADVANCE}}$ request; all others are **proper** $\boxed{\text{ADVANCE}}$ requests.

A $\boxed{\text{NULLIFY}}$ request will usually stop and make null and void any daton evaluation which may have taken place after the last $\boxed{\text{ADVANCE}}$ (or after ***initialisation***, if there has been no $\boxed{\text{ADVANCE}}$ yet). If a $\boxed{\text{COMPUTE}}$ request follows directly after the $\boxed{\text{NULLIFY}}$ (i.e. without an $\boxed{\text{ADVANCE}}$ in between) the inferior will set out to compute the value of the same daton as for the previous $\boxed{\text{COMPUTE}}$ request. A ***single-outport*** $\boxed{\text{COPY}}$ node may be inserted in the arc wherever the re-computation of intermediary results shall be avoided.

An actively computing inferior may in turn have issued a subordinate $\boxed{\text{COMPUTE}}$ request (i.e. it is a subordinate ***superior***). If such a sub-superior gets a $\boxed{\text{NULLIFY}}$ request, it will halt its current computation, do the necessary clear-up (like propagating the $\boxed{\text{NULLIFY}}$ request to the sub-inferiors), and it will then await the next request.

Most inferiors have inports. If such an inferior gets an $\boxed{\text{ADVANCE}}$ request, it will first do the same as in a $\boxed{\text{NULLIFY}}$ request, it will then propagate the $\boxed{\text{ADVANCE}}$ request to the inports, and it will increment its own index counter by one. It will finally await the next request, i.e. it will enter the dormant state.

Request Propagation

Two diametrically opposed strategies govern the ***propagation*** of requests, though both aim towards efficiency. (These request propagation strategies are also reflected in the priority scheduling, ↑ 4.7.)

$\boxed{\text{COMPUTE}}$ requests cause daton evaluations, and daton evaluations tend to be ***expensive***. $\boxed{\text{COMPUTE}}$ requests are therefore issued as ***sparingly*** as possible, and they are withdrawn (by $\boxed{\text{NULLIFY}}$) as soon as it becomes certain that the evaluation result is not needed.

Exceptions, on the other hand, are propagated at the ***earliest possible*** moment. We do so because, in general, exceptions are capable of ***releasing computing resources*** further upstream. Exceptions usually trigger some administration, but even that is considered to be "well spent". ***Exceptions must never cause infinite looping or*** $\boxed{\text{COMPUTE}}$ ***requests***. Care must be taken in the ***act design*** to ensure that this rule is not violated. This is not always trivial; for example, computations can be accidentally caused if a bare $\boxed{\text{ADVANCE}}$ is issued to a poorly designed $\boxed{\text{WVR}}$ actor.

Closing Remarks

Various other protocols were tried out, and the above design proved best for implementation. Among the worst of the alternatives was the one which combined $\boxed{\text{COMPUTE}}$ and $\boxed{\text{ADVANCE}}$ into a ***single*** request (↑ beginning of 4.2). In order to permit $\boxed{\text{NULLIFY}}$ requests in that design, even the simplest actor had to be provided with memory in which computed values could be saved.

Node actor ***initialisation*** is part of the protocol, in the wider sense. We chose, however, to describe actor initialisation in connection with actor creation in section 4.3.1 (B).

4.3 The Translation Proper

This section presents the method for translating any Lucid graph into its LUX equivalent, namely a net of initialised node actors. This side of the translation algorithm is independent from the particular design of the node acts. (Our quiet assumption of ***demand driven*** evaluation, though, has a certain bearing on this section.) We pretend for the remainder of this section that a suitable act has already been defined for each node type. There is no danger that this assumption leads us into a vicious circle.

Exceptions, on the other hand, are propagated at the ***earliest possible*** moment. We do so because, in general, exceptions are capable of ***releasing computing resources*** further upstream. Exceptions usually trigger some administration, but even that is considered to be "well spent". ***Exceptions must never cause infinite looping or*** $\boxed{\text{COMPUTE}}$ ***requests***. Care must be taken in the ***act design*** to ensure that this rule is not violated. This is not always trivial; for example, computations can be accidentally caused if a bare $\boxed{\text{ADVANCE}}$ is issued to a poorly designed $\boxed{\text{WVR}}$ actor.

Closing Remarks

Various other protocols were tried out, and the above design proved best for implementation. Among the worst of the alternatives was the one which combined $\boxed{\text{COMPUTE}}$ and $\boxed{\text{ADVANCE}}$ into a ***single*** request (↑ beginning of 4.2). In order to permit $\boxed{\text{NULLIFY}}$ requests in that design, even the simplest actor had to be provided with memory in which computed values could be saved.

Node actor ***initialisation*** is part of the protocol, in the wider sense. We chose, however, to describe actor initialisation in connection with actor creation in section 4.3.1 (B).

4.3 The Translation Proper

This section presents the method for translating any Lucid graph into its LUX equivalent, namely a net of initialised node actors. This side of the translation algorithm is independent from the particular design of the node acts. (Our quiet assumption of ***demand driven*** evaluation, though, has a certain bearing on this section.) We pretend for the remainder of this section that a suitable act has already been defined for each node type. There is no danger that this assumption leads us into a vicious circle.

Every Graph Lucid program consists of nodes and arcs, and its translation can correspondingly be described in two parts:

(A) the translation of the nodes and

(B) the translation of the arcs.

First (section 4.3.1) we are going to present the translation algorithm for programs ***without*** recursive UDFs. Before progressing to an algorithm for programs with recursive UDFs (section 4.3.3) we will study UDFs and related topics (section 4.3.2).

4.3.1 Programs without Recursive UDFs

Let us first deal with the translation of particularly simple Lucid programs, namely those without recursive UDFs. More precisely, this section describes only the translation of programs without UDFs altogether. However, section 4.3.2 will show how to remove non–recursive UDFs (viz. UDF expansion), a process which can be easily carried out before applying the algorithm of this section.

Under this restriction the nodes in the Lucid graph can be *labelled* with natural numbers, with a known finite bound (see also Fibonacci example, two pages below). The root act establishes the LUX counterpart for the graph by **(A)** first creating exactly one actor for each individual node in the graph. The choice of act is determined by the node type, of course. While the root actor creates the actors (in the sequence of the labelling number) it enters the name of each new actor into a table. $\boxed{\text{COPY}}$ node actors (↑ 4.6) are special in having a separate actor name for each outport (1, 9 and 10 in the Fibonacci example), in addition to the name of the $\boxed{\text{COPY}}$ node actor itself (inport, labelled 1 in the example). Immediately after creating a $\boxed{\text{COPY}}$ node actor, the creator gets back from that actor a few messages, each telling the name of one $\boxed{\text{COPY}}$ outport actor.

Every Graph Lucid program consists of nodes and arcs, and its translation can correspondingly be described in two parts:

(A) the translation of the nodes and

(B) the translation of the arcs.

First (section 4.3.1) we are going to present the translation algorithm for programs ***without*** recursive UDFs. Before progressing to an algorithm for programs with recursive UDFs (section 4.3.3) we will study UDFs and related topics (section 4.3.2).

4.3.1 Programs without Recursive UDFs

Let us first deal with the translation of particularly simple Lucid programs, namely those without recursive UDFs. More precisely, this section describes only the translation of programs without UDFs altogether. However, section 4.3.2 will show how to remove non–recursive UDFs (viz. UDF expansion), a process which can be easily carried out before applying the algorithm of this section.

Under this restriction the nodes in the Lucid graph can be ***labelled*** with natural numbers, with a known finite bound (see also Fibonacci example, two pages below). The root act establishes the LUX counterpart for the graph by **(A)** first creating exactly one actor for each individual node in the graph. The choice of act is determined by the node type, of course. While the root actor creates the actors (in the sequence of the labelling number) it enters the name of each new actor into a table. $\boxed{\text{COPY}}$ node actors († 4.6) are special in having a separate actor name for each outport (1, 9 and 10 in the Fibonacci example), in addition to the name of the $\boxed{\text{COPY}}$ node actor itself (inport, labelled 1 in the example). Immediately after creating a $\boxed{\text{COPY}}$ node actor, the creator gets back from that actor a few messages, each telling the name of one $\boxed{\text{COPY}}$ outport actor.

In the graph, arcs ***connect*** the nodes. Correspondingly, there must be connections between the node actors. After the creation of all the actors, the root actor establishes these connections in **(B)** the ***initialisation*** of all the actors. It informs each actor of the names of the actors ***at its inports*** (i.e. the node actors which produce the operand daton values). Since we deal with a demand driven implementation, each actor takes a ***dominant*** role over the actors at its inports and it takes a ***servile*** role with regard to the actor at its outport. Operand actors are therefore called **inferiors**, and the requesting actor is called the **superior**. At the program start, each actor needs to know only the names of its ***inferiors***.

In the translation stage **(B)**, an initialisation message with the names of the inferiors is sent to each node actor. The initialisation message is the sequence:

<DATON, $name_0$, $name_1$, $name_2$, . . .>

Each $name_i$ appears at the index position corresponding to its inport subscript i. The component DATON is due to our message convention (↑3.2.1). We use the convention that actors for nodes with ***no*** inport (constant and READ nodes) get ***no*** initialisation. The WRITE node must be the ***last*** to be initialised, this makes sure that requests are not sent to nodes which are still waiting to be initialised. The reason lies in WRITE being top in the request hierarchy.

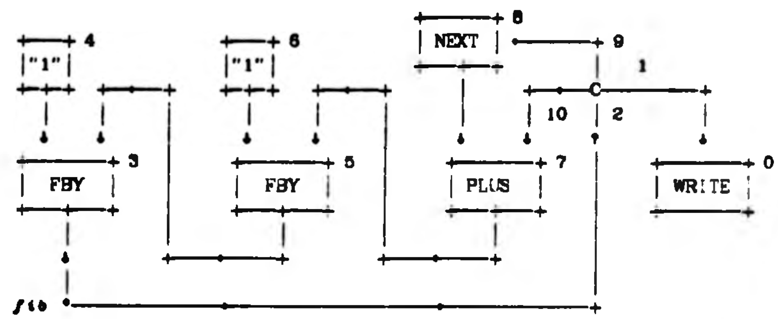
It has been said before that every node actor can be initialised only by its ***creator***. In our special case (all UDFs fully expanded) the ***root actor*** is the creator of all node actors.

Example (Fibonacci)

Let us apply these rules to a simple example program (↑ chapter I) which computes the Fibonacci series:

```
fib  WHERE
      fib  = 1 FBY ( 1 FBY ( (NEXT fib) + fib ) )   ;
     END
```

Here is its corresponding graph, with the nodes labelled by numbers (• 2.2 and 4.3.1):



Every Lucid program is an expression which yields a result (here *fib*), and this result flows obviously into a $\boxed{\text{WRITE}}$ node. Here is the $\boxed{\text{Act\_Root}}$ which would generate the net of actors:

```
ACT Act_Root_ ;                       (* Root act for Fibonacci example.  *)
    VAR
       node : ARRAY [0..10] OF ACTOR ;

    (* Furthermore, there must be ACT declarations for:                  *)
    (* Constant, NEXT, COPY, FBY, PLUS and WRITE.                        *)

    BEGIN                             (* Act_Root_ has no initialisation. *)
        node[0] := CREATE (Act_Write_, "console") ;
        node[2] := CREATE (Act_Copy_ , 3) ;           (* 3 outports *)
        ( , , node [1]) := RECEIVE FROM (node[2]) ;
        ( , , node [9]) := RECEIVE FROM (node[2]) ;
        ( , , node[10]) := RECEIVE FROM (node[2]) ;
        node[3] := CREATE (Act_Fby_     ) ;
        node[4] := CREATE (Act_Const_, 1) ;
        node[5] := CREATE (Act_Fby_     ) ;
        node[6] := CREATE (Act_Const_, 1) ;
        node[7] := CREATE (Act_Plus_    ) ;
        node[8] := CREATE (Act_Next_    ) ;
        Set_Priority (node[0], top_priority) ;

        SEND (DATON, node[9])            TO (node[8]) ;
        SEND (DATON, node[8], node[10])  TO (node[7]) ;
        SEND (DATON, node[6], node [7])  TO (node[5]) ;
        SEND (DATON, node[4], node [5])  TO (node[3]) ;
        SEND (DATON, node[3])            TO (node[2]) ;
        SEND (DATON, node[1])            TO (node[0]) ;
    END ;
```

Act_Root_ has no exception part, since it gets no exceptions. The root actor terminates itself. After the root actor is gone, all the "driving force" for computations will emanate from the WRITE actor, node[0].

The Node Numbering Rule

You may have guessed that the numbering of the nodes follows not just a whim but a rule, yet to be explained. To begin with, the lowest label numbers are given to the nodes which generate the ultimate ***driving force*** for computation. We deal here with ***demand*** driven DF, and we attach label 0 to our WRITE node, the ultimate demander. Many nodes force other nodes into action. In demand driven DF, nodes tend to propagate requests to the nodes at their inports, and thus the driving force **flows upstream**. We number the nodes in such a way that ***every node requests only from nodes with higher label numbers***. The node numbers increase therefore in the upstream direction. Nodes are created and initialised in the order of ***decreasing*** label number. – This numbering rule caters even for subnets with many outports, and can be adapted for input driven DF. The numbering rule ensures that each operand actor is itself readily initialised before its name is passed around to other actors (consequence: requests cannot be sent to yet uninitialised actors).

Outlook

Scheduling and priorities will be discussed in section 4.7. They are indispensable for correct and efficient program execution.

It is advisable in the first reading pass to skip the remainder of this section 4.3, to continue this chapter from 4.4, and then to re–read the entire chapter without omissions. – Before we can go on to Lucid programs in general, a review of UDFs and subnets is in place.

4.3.2 Abstraction and Expansion (UDFs and Subnets)

4.3.2.1 A+E in Equational Lucid

Abstraction lies at the root of many programming techniques. UDFs, subnets (↑ 2.2) in Graph Lucid, and subnets of actors are the kind of abstractions which interest us here. For any particular abstraction there is always one **definition** and an arbitrary number of **references**. References are just the means for making use of definitions. The definition of abstraction *XYZ* states "here is the shape of the object you may substitute for the reference if you want to obtain a result from *XYZ*". The **Rewrite Rule** characterises the meaning of abstractions more precisely: if we take any structure *S*, and substitute in *S* each reference to *XYZ* by the object specified in the definition of *XYZ*, the outcome *S'* will behave the same as the original structure *S*. – In the abstraction, the **formal** operands, if there are any, stand as symbols (place holders) for the **actual** operands quoted in each reference. It is common to call the ensemble of the actual operands the **environment** of the reference (remember we have eliminated all global variables).

The actual replacing of the reference by its ***true essence*** (as given in the ***definition***) is called **expansion**. In the expansion, each occurrence of a formal operand is substituted by its corresponding actual operand. We will see that, in some situations, a high degree of abstraction is favoured while in some other situations one should aim for expansion.

Here is an example of a UDF ($\boxed{cf}$, $\boxed{xf}$ and $\boxed{yf}$ are *F*ormal operands whereas $\boxed{ca}$, $\boxed{sa}$ and $\boxed{ta}$ are *A*ctual operands):

```
// definition:
  Mymerge (cf,xf,yf)  =  IF cf THEN   xf UPON cf
                              ELSE   yf UPON NOT cf  FI ;

// reference (assume p, sa and ta have been defined elsewhere):
  m  =  Mymerge (0.5 < p,  sa,  ta) ;
```

Expansion of this UDF reference yields:

```
  ca  =  0.5 < p;
  m   =  IF ca THEN  sa UPON ca
               ELSE  ta UPON NOT ca  FI ;
```

Abstraction is promoted in Software Engineering since it makes programs easier to understand and maintain. Whenever we analyse any substantial Lucid program, we are almost bound to find particular ***substructures*** re–occurring in many places, the more so if we make provisions for minor variations. As we know from Software Engineering, this is almost unavoidable with any substantial program. We are advised to formulate one abstraction for each substructure, and to replace each instance of the substructure by a reference to that abstraction. Software engineering teaches, furthermore, that it is a good idea to subdivide (to "***structure***") programs into ***purpose related units***, and to abstract each unit.

Subexpressions and UDFs

Our translation algorithm presupposes that the Lucid program is in ***monomeric*** form (↑ 2.1.6 and 4.3.3.1): there is at most ***one*** operator in each definition. Each UDF is an operator, and the monomeric form permits only variables and constants as actual operands, i.e. only ultra–simple expressions are allowed. We will come back to this point later on (↑ 4.3.3.2, ***Making Subexpressions into UDFs***).

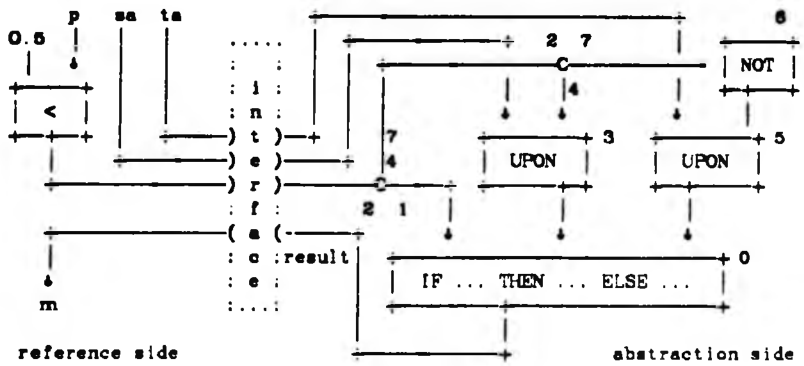
4.3.2.2 A+E in Graph Lucid

All this applies equally to Graph Lucid, since Graph Lucid is a bijection of equational Lucid. Like programs, any Lucid graph can be subdivided into segments. Each of these segments is a ***subnet*** (↑ 2.2). Again, there will often be great similarity among the subnets. This suggests the definition of ***classes*** (=abstractions) of subnets. Subnet classes are the exact counterpart for UDFs. We use in the following ***subnet*** often in the meaning of ***subnet class***.

Every UDF node represents two kinds of structure, and its great power results from its mediating between the two. Its ***outside structure*** is that of a single node (the UDF node), while its ***inside structure*** reveals a subnet composed of numerous nodes.

Each subnet has ***open*** arcs, i.e. it has outport (and inport) arcs which are not connected to any node, but instead are connected to an ***interface***. Such an **interface** is a combined array of ***plugs*** (open inport arcs) and ***sockets*** (open outport arcs) which will eventually link up with complementary sockets and plugs. The inports and outports of every subnet reference must match the requirements of its abstraction, so that plugs and sockets can be paired.

Here is the Mymerge example from above, this time as a Lucid graph:



(Ignore the numbering of the subnet nodes, for the time being.) On the left is the UDF reference, and on the right we see an instance of the subnet for Mymerge, both connected by an interface. The picture is a snapshot of the state of affairs when expansion is half complete. Before the expansion, the subnet on the right is only conceptually present, symbolised by a Mymerge node. At least in some implementations, expansion goes one step further than shown above: it replaces the interface by direct through connections (↑ 6.3, operand redirection).

Each UDF reference divides the Lucid graph into *two* subnets, the subnet for the abstracted side, and the subnet for the referencing side.

Abstraction and expansion have counterparts in subnets of actors, and with the aid of these counterparts even programs with recursive UDF's can be implemented in LUX.

4.3.3 Application of Abstraction and Expansion in LUX

4.3.3.1 Programs with Recursive UDFs

Lucid programs with recursive UDFs are only slightly more complicated to translate than the simple programs considered in section 4.3.1.

Example (Sieve): Lucid program and graph

The prime numbers can be computed by an algorithm known as the "Sieve of Eratosthenes", and this algorithm can be elegantly described by a Lucid program with a recursive UDF (original program due to Gilles Kahn). We start here with the Lucid program, we will present all the translation steps, and we will present all the various acts required for it, including the translation program. In chapter V, the dynamics of program execution will be illustrated, using the Sieve program as the example. Here is its Lucid program:

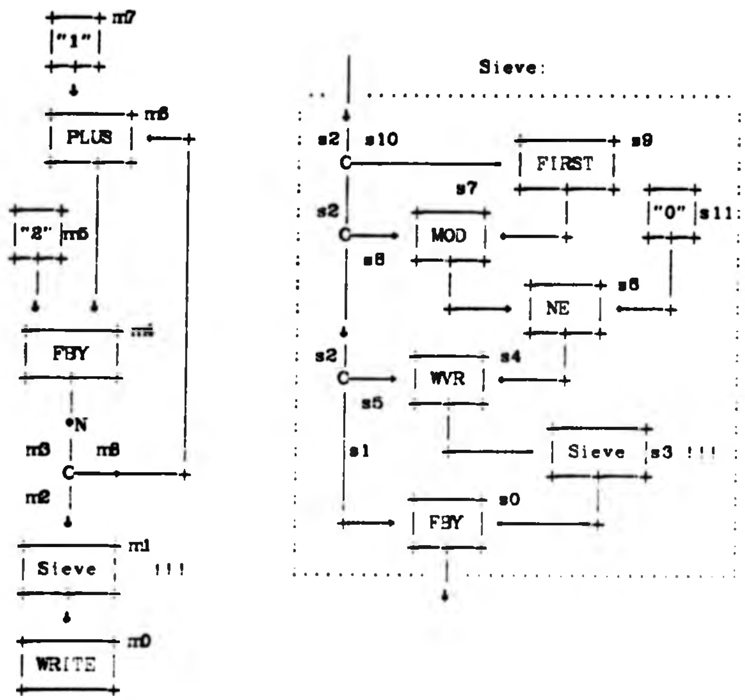
```
Sieve(N)
  WHERE
    N          =  2 FBY N+1   ;
    Sieve(i)   =  i   FBY
                  Sieve (i WVR ((i MOD FIRST i) NE 0)) ;
  END
```

Let us make the program monomeric:

```
m1  WHERE
     Sieve(i)   = s0  WHERE
                       (s1,s5,s8,s10)   =  COPY (i) ;
                       s9   =      FIRST s10  ;
                       s7   =  s8  MOD     s9  ;
                       s6   =  s7  NE       0  ;
                       s4   =  s5  WVR     s6  ;
                       s3   =      Sieve (s4)  ;
                       s0   =  s1  FBY     s3  ;
                      END ;

     m6          =  m8  +       1  ;
     N           =  2   FBY    m6  ;
     (m2, m8)    =      COPY  (N ) ;
     m1          =      Sieve (m2) ;
    END
```

It is now quite easy to generate the corresponding Lucid graphs, with labelled nodes (the labels in the *main program* have been prefixed m, and those in the Sieve have been prefixed s):



The Finite Program vs. the Unbounded Net

The graph of the *main program* on the left contains a reference to the UDF Sieve, and the graph of Sieve itself, right, contains a further reference to Sieve. Any reference to a UDF is treated the same as the reference to any operator. The only difference is that there must be a definition for each UDF, whereas all other operators are readily defined.

The graph on the right reveals the true nature of the UDF. Outwardly it is just a node, but inside it contains a whole subnet. This subnet comprises another reference

to $\boxed{\text{Sieve}}$, which symbolises a further subnet. The program specifies effectively an ***infinite*** nesting of UDFs (in Graph Lucid terms: an infinite net), rather like:

```
mainprog (
  ... Sieve (
    ... Sieve (
      ... Sieve (
        ... Sieve (
              and so on ad infinitum
                  ) ...
                ) ...
              ) ...
            ) ...
          )
```

Lucid programs can be analysed in a rather static ("***denotational***") manner. However, when we discuss their execution, we cannot avoid thinking in terms of execution time (***operationally***, dynamically). Programs are executed in a succession of fundamental operations, computation steps. In this thesis, we call $\boxed{\text{FBY}}$, $\boxed{\text{NEXT}}$, $\boxed{\text{IF–THEN–ELSE}}$ and the usual pointwise operators (addition etc.) **primitive operators**; more about them in section 4.4. In LUX, the **fundamental operators** are $\boxed{\text{CREATE}}$, $\boxed{\text{SEND}}$, $\boxed{\text{RECEIVE}}$, $\boxed{\text{EXCEPTION}}$, the system functions, the primitive operators, but not UDFs. $\boxed{\text{FIRST}}$, $\boxed{\text{UPON}}$ and $\boxed{\text{WVR}}$ are counted as UDFs.

Delayed Net Expansion

Every abstraction reference needs to be expanded (into a set of actors) before it can truly take part in a computation. However, if all expansion had to be carried out at the start of program execution, a disaster might occur, since every reference to a recursive UDF would generate ***infinitely*** many actors. The ***size*** of a net with recursive UDFs can not be pre-determined in general, it may even be ***unbounded***.

This size problem can be resolved by ***delaying*** the UDF expansion. During program execution, there is for every instruction (and that includes any UDF reference) a moment where it is used for the first time. In demand driven evaluation, this moment is the one ***where the first request arrives***. (For some instructions this moment may never arrive.) A request, directed to the UDF, can be serviced only by the expanded UDF, but expansion can be delayed up to this moment. Up to that moment, the abstraction is kept in a *preliminary* state where the actor subnet has not actually been expanded, although all the information necessary for expansion is at hand (i.e. actor initialisation complete). This method has the attraction that only *finitely* many actors exist at any moment.

If expansion is delayed up to the last moment, we speak of a **lazy expansion**. Its obvious opposite is **eager expansion**, where the subnet is expanded a good while before its first use. The extreme of eager expansion is the expansion before the start of program execution (↑ 4.3.1); this is called **static expansion**. We will come back to eager and lazy expansion when we discuss act expansion (↑ 6.2).

UDF Acts

UDF acts are the LUX counterpart for UDFs. Every single UDF actor (outside structure) stands for a subnet of actors (inside structure). UDF references (code for issuing requests) have the same form as any other node actor reference, since we agreed on a uniform protocol.

In the framework of node actors, the word *abstraction* means "yet unexpanded subnet of actors", and every UDF actor has therefore two states (similar to a finite state machine):

- the *abstracted* state (the preliminary state), and
- the *expanded* state (the state during execution).

Speaking in implementation terms, every UDF actor contains, right after its own initialisation, code which **(A)** creates all the actors in the subnet and then **(B)** initialises them. Both **(A)** and **(B)** are carried out very much in the way described in sections 4.1 and 4.3.1, but with the difference that now the UDF actor is the creator and initialiser.

Example ([Sieve]): UDF act

Here is the [Act_Sieve] which would generate the appropriate actor subnet (node numbers same as in the graph):

```
ACT Act_Sieve ;                      (* Act for UDF Sieve.              *)
    LABEL 1 ;
    VAR
        node   : ARRAY [0..11] OF ACTOR ;
        inport : ARRAY [0..0]  OF ACTOR ;
        skip   : INTEGER ;

    (* Furthermore, there must be ACT declarations for.                 *)
    (* COPY, NOT, UPON and IF.                                          *)

    BEGIN        (* Initialisation:                                     *)
        skip := 0 ;
        ( , , inport[0]) := RECEIVE FROM (Creator) ,

        (* Due to its low intrinsic priority († 4.7) the node           *)
        (* actor will wait here until the first request arrives.        *)

        (* Below it will be shown that some further code must           *)
        (* be inserted here (intercepting ADVANCE exceptions)           *)

        (* The X-part:                                                  *)
        node [0] := CREATE (Act_Fby_     ) ;
        node [2] := CREATE (Act_Copy_,  4) ,   (* 4 outports            *)
        ( , , node [1]) := RECEIVE FROM (node[2]) ;
        ( , , node [5]) := RECEIVE FROM (node[2]) ,
        ( , , node [8]) := RECEIVE FROM (node[2]) ;
        ( , , node[10]) := RECEIVE FROM (node[2]) ;
        node [3] := CREATE (Act_Sieve    ) ;   (* the recursion         *)
        node [4] := CREATE (Act_Wvr_     ) ;
        node [6] := CREATE (Act_Ne_      ) ;
        node [7] := CREATE (Act_Mod_     ) ;
        node [9] := CREATE (Act_First_   ) ;
        node[11] := CREATE (Act_Const_, 0) ;

        SEND (DATON,   node[10])             TO (node [9]) ;
        SEND (DATON,   node [8], node [9])   TO (node [7]) ;
        SEND (DATON,   node [7], node[11])   TO (node [6]) ;
        SEND (DATON,   node [5], node [6])   TO (node [4]) ;
        SEND (DATON,   node [4])             TO (node [3]) ;
        SEND (DATON, inport [0])             TO (node [2]) ,
        SEND (DATON,   node [1], node [3])   TO (node [0]) ;
                                                                     :1
1:      Pass_Through (node[0], skip) ;
    END ;
```

Every UDF act uses the procedure Pass_Through. This procedure contains the Y–part, and it passes all requests on to node[0], the highest ranking actor within the subnet, and conversely, it passes all replies back to the superior of the UDF actor.

```
PROCEDURE Pass_Through (node0 : ACTOR; skip : INTEGER) ;
    LABEL 1 ;
    VAR
        superior : ACTOR   ; request : MSGTYPE ;
        reply    : ANYTYPE ; index   : INTEGER ;
    BEGIN
      FOR index := 1 TO skip
      DO  EXCEPTION (ADVANCE, index) TO (node0) ;

      REPEAT
        WHILE TRUE
        DO BEGIN                                                  :1
             (superior, request, index) := RECEIVE () ;           :1

             (*  The Y-part:                                  *)  :1
             reply := GetDaton (index, node0) ;                   :1
             SEND (DATON, reply) TO (superior) ;
           END ;

        (* Exception part:                                    *)
1:      (request, index)) := Reveal ;
        IF    request = ADVANCE
        THEN  EXCEPTION (request, index) TO (node0) ;
        RESET ;
      UNTIL FALSE ;
    END ;
```

Act_Sieve begins with the initialisation of the actor itself. The formal operand I from the Lucid program translates thus into a storage cell which the creator fills with the name of the actual operand actor. This is followed (X–part) by the expansion proper, the creation and initialisation of the subnet actors. The act ends with a call of the procedure Pass_Through, which contains its Y–part.

The X–part resembles clearly the Act_Root from the Fibonacci program (↑ 4.3.1). While scheduling will be properly discussed in section 4.7, we briefly mention here that all node actors (other than WRITE) have initially an extremely low scheduling priority. Execution of the X–part of any actor starts only upon arrival of the first request. In the case of UDF actors, this makes sure that the subnet is created not earlier than really necessary.

The call of the $\boxed{\text{Pass\_Through}}$ procedure is ***eternal***, i.e. the procedure is called once, and, because of its eternal loop, there is no return from it. The essential part of the procedure, the eternal loop, has been copied straight from the identity node (simply remove the scaling from the $\boxed{\text{Act\_Scale}}$, ↑ 3.4.4). Since $\boxed{\text{Pass\_Through}}$ contains no computation it is a prime target for optimisation, and we shall indeed discuss ***expansion of a UDF reference*** (↑ 6.2), optimisation of recursive UDFs by ***tail recursion*** (↑ 6.6), and ***operand redirection*** (↑ 6.3), all of which are applicable here.

Doors need not be provided in the subnet expansion code ($\boxed{\text{CREATE}}$ and initialise) since the superior will be hung in its first request (the one which caused the expansion) and can therefore not issue a further request during expansion. (One might consider this approach as crude and replace it by one which has a request $\boxed{\text{RECEIVE}}$ before the expansion code. Such a refined version would indeed need doors.)

The node numbering rule (from the $\boxed{\text{root\_actor}}$, ↑ end of 4.3.1) extends unchanged to UDF actors. Since that rule has certainly been adhered to during the initialisation of the UDF actor itself, all subnet inports (actors for actual operands) can be assumed to be ready for use.

Initial $\boxed{\text{ADVANCE}}$ Requests

For safety, a piece of extra code must be inserted between initialisation and X–part:

```
WHILE    Reveal = ADVANCE
DO BEGIN
     (request, index) := Reveal ;
     IF   index = finalindex
     THEN EXCEPTION (request, index)
          TO (inport[0], ... inport[n])
     ELSE skip := skip + 1 ;
     RESET ;
   END ;
```

The cell $\boxed{\text{skip}}$ adds up any bare $\boxed{\text{ADVANCE}}$ requests initially sent to the UDF. Only the first $\boxed{\text{COMPUTE}}$ request will cause the UDF expansion. $\boxed{\text{ADVANCE}}$ requests must never cause "expensive" actions, such as the UDF expansion. (Without $\boxed{\text{skip}}$ it would be impossible to implement a UDF like WVR.)

If the first request ever to be sent to the UDF is an ADVANCE, finalindex, the request is propagated to the operand actors and the subnet creation is suppressed. Without this extra code, recursive UDFs would be liable to deadlock: if ADVANCE, finalindex was the first request issued to such a UDF, its actor would settle down to building and inactivating subnets forever. This matter will be understood more easily once the FBY act has been explained (↑ 4.5.6). A more radical approach to the whole finalindex problem will be presented in 6.3 (the KILL request).

Example (Sieve): root act

Here is the Act_Root_ which would generate the *main program* for Sieve:

```
ACT Act_Root_ ;                          (* Root act for Sieve example.  *)
    VAR
       node : ARRAY [0..8] OF ACTOR ;
    (* Furthermore, there must be ACT declarations for.           *)
    (* Constant, COPY, FBY, PLUS and WRITE.                       *)

    BEGIN                    (* Act_Root_ has no initialisation.  *)
        node [0] := CREATE (Act_Write_, "console") ,
        node [1] := CREATE (Act_Sieve    ) ;
        node [3] := CREATE (Act_Copy_ , 2) ;     (* 2 outports     *)
        ( , , node[2]) := RECEIVE FROM (node[3]) ;
        ( , , node[8]) := RECEIVE FROM (node[3]) ;
        node [4] := CREATE (Act_Fby_     ) ;
        node [5] := CREATE (Act_Const_, 2) ;
        node [6] := CREATE (Act_Plus_    ) ;
        node [7] := CREATE (Act_Const_, 1) ;
        Set_Priority (node[0], top_priority) ,

        SEND (DATON, node[7], node[8]) TO (node [6]) ;
        SEND (DATON, node[5], node[6]) TO (node [4]) ;
        SEND (DATON, node[4])          TO (node [3]) ;
        SEND (DATON, node[2])          TO (node [1]) ,
        SEND (DATON, node[1])          TO (node [0]) ;
    END ;
```

Interlude

UDFs, subnets in Lucid graphs, and subnets of initialised actors correspond so closely to each other that most generalisations about either apply to all three. When looking at the figure in 4.3.3.1 one is tempted to believe that every instance of $\boxed{\text{Sieve}}$ is just a "carbon copy" of the UDF $\boxed{\text{Sieve}}$. This view is quite in harmony with the functionality definition ("replacing the UDF reference by the UDF definiens does not change the computation result"). But the carbon copy approach cannot be generalised to cover ***operational*** objects, like actors. Many node actors have memory. An abstraction, on the other hand, can not contain memory but can at best contain information ***where*** to allocate storage space, and ***how much***. In the operational interpretation of DF Lucid graphs, there is a silent understanding that each arc has initially an empty queue associated.

When implementing recursive UDFs, delayed expansion is the method to choose. However, implementation of recursive UDFs is merely one application of delayed expansion. Let us take a short look at the general application area.

4.3.3.2 Further Applications of A+E in LUX

Above, in section 4.3.2, we outlined the reasons for abstraction from the Software Engineering point of view. Quite separately, abstraction offers also advantages to system implementors. They are attracted by its particularly economical use of storage space: only one copy of the UDF definiens needs to be held in store, and no actor space is claimed until the first $\boxed{\text{COMPUTE}}$ occurs. Abstraction has one inherent disadvantage: its use incurs some extra ***administration*** cost, and this penalty re-applies normally to each daton evaluation.

For the execution of some Lucid program fragments (subnets) the prediction can be made that they will go through a ***protracted initial period of inactivity***. Store is used very economically if during this period the subnet is kept in abstracted form.

An optimising compiler might detect such subnets through program analysis. The above property applies particularly often to actual operand expressions of UDFs. In many implementations, efficiency is improved by abstracting all but the simplest (i.e. variables or constants) subnets with the above property.

The author admits freely not to know a universal rule for identifying all subnets which have such a "protracted initial period of inactivity". Only a few prominent instances will be presented in this thesis, namely recursive UDFs (↑ 4.3.3.1), inactive subnets, and $\boxed{\text{IF}}$ with constant condition (↑ 6.6).

The optimising compiler may contain a device for expanding some of the program writer's abstractions, but it may also contain a device for introducing abstractions of its own making. For the remainder of this section we will, however, assume that we are not using such an optimising compiler. Suggestions for optimisation can be found in chapter VI.

There is a certain limit, a minimal UDF complexity, from where on abstraction has only disadvantages, both in execution speed and storage. UDF expansion is indicated if the UDF definiens contains no operator ($f(x)=x$), and also if it has merely one operator and is non-recursive ($f(x,y)=x+y$). References to such ultra-simple UDFs can be eliminated by the compiler.

Making Subexpressions into UDFs

Any expression is only as likely to be used as the structure that refers to it. If this structure is itself inactive for a protracted initial period, it may be advisable to make the expression into a UDF.

For example, an actual operand expression of a UDF is certainly never used before the UDF itself, and abstraction of the operand expression may be indicated. – Similarly, program fragments like the following are not uncommon in Lucid programs:

```
a = IF    FIRST c
    THEN  (x+3) * x
    ELSE  1 / (1-x)   FI ;
```

The $\boxed{\text{IF}}$ condition is evaluated once, and it is constant (* 6.6). This condition selects either the $\boxed{\text{THEN}}$ operand or the $\boxed{\text{ELSE}}$ operand, and the other operand will never be used. The code for this operand will forever idly waste store. However, the example can be rewritten into:

```
ThenFunc (x) = (x+3) * x ;
ElseFunc (x) = 1 / (1-x) ;

a = IF   FIRST c
    THEN ThenFunc (x)
    ELSE ElseFunc (x) FI ;
```

This has given us two extra UDFs, $\boxed{\text{ThenFunc}}$ and $\boxed{\text{ElseFunc}}$, the abstractions of the original expressions. Only the unexpanded *UDF* actors (i.e. not their subnets) are created together with the $\boxed{\text{IF}}$ actor, and only either of them will ever be expanded.

4.3.4 Summary of Translation Proper

We present the algorithm once more, this time in *imperative* form. The program is first put into a more convenient form through a few transformations:

(a) We make the Lucid program ***monomeric***.

(b) A ***cross-reference*** is generated, covering all identifiers in the Lucid program (***simple*** as well as ***function*** definitions). The transitive closure of this cross-reference is generated. All definitions which are not in the transitive closure of the ***program result*** can be deleted. ***Recursive*** function definitions can now be marked as such. (Recursively defined variables constitute ***cycles***, * 6.1.)

(c) We replace all instances of $\boxed{\text{FIRST}}$, $\boxed{\text{UPON}}$, $\boxed{\text{WVR}}$ and $\boxed{\text{ASA}}$ by their UDF equivalents (↑ 4.5). Furthermore, we substitute all instances of ***currenting*** by suitable $\boxed{\text{Igloo}}$ functions (↑ appendix B).

(d) Through the cross-reference we can locate all occurrences of ***global*** variables, and we eliminate them by converting them into extra UDF operands. After this elimination, UDFs acquire all datons as UDF operands and deliver them as UDF results. As a result, the entire program consists of completely separate ***segments***, namely one ***main program*** (the subnet which contains the $\boxed{\text{WRITE}}$ node) and any number of UDFs.

(e) Sizeable UDFs should not be expanded ***eagerly*** if they have more than ***one*** reference, including self–references of recursive UDFs. There is ***no*** law forbidding the textual expansion of UDFs with only one reference. We may now expand certain undesirable UDFs. Conversely, some complicated reason may persuade us to introduce some new UDFs (↑ 4.3.3.1 and 6.2).

(f) All ***multiple*** references to a variable must be resolved by $\boxed{\text{COPY}}$ nodes.

The Translation Strategy

We apply the translation program proper first to the Lucid "main program" and then in turn to each UDF. The translation program incorporates the ***node numbering rule*** from section 4.3.1.

Every net or subnet contains one ***highest ranking*** node. For the "main program" this is the $\boxed{\text{WRITE}}$ node, while for any UDF this is the node which computes the very UDF result. According to the Lucid syntax, every program or UDF is an expression, and there is therefore only one highest ranking node per UDF or per main program. In order to translate UDFs correctly we must remember that even each formal operand maps into a ***node actor*** which computes that operand. The translation becomes easier if we substitute each formal operand by a subscripted dummy variable $\boxed{\text{node}[\rightarrow i]}$ (with i ranging over the inport numbers *1, 2 ... n*).

In the following we analyse Lucid graphs ***recursively***. We start by looking at the highest ranking node, but before looking at a node itself, we look first at the producers of its operands (these will be lower ranking node actors). In Lucid graphs, the arrows indicate the direction of flow of datons. Effectively, we make excursions ***upstream*** along the arcs, and we generate code on the "return travel" ***downstream***. In the course of this process, a number will be attached to each node, and code for creation and initialisation of the corresponding actor will be generated. It is obvious that this translation process terminates (i.e. no further recursion) when encountering the following operators:

- an operator with no operands (constants, $\boxed{\text{READ}}$),
- UDF inports, or
- any $\boxed{\text{COPY}}$ node which has already been translated.

Each $\boxed{\text{COPY}}$ node delivers operands to ***many*** other nodes, and it will therefore be reached repeatedly in our translation algorithm. But of course, code must be generated for each $\boxed{\text{COPY}}$ node only ***once***. This can be achieved by attaching a Boolean flag to each $\boxed{\text{COPY}}$ node.

Representation for Graph Lucid

Below we will render the translation algorithm as a PASCAL program, which has been implemented and properly tested (↑ appendix C). The program presupposes that the Lucid graph is pregiven, the outcome of the transformations (a) ... (f) just described. The graph is built up from PASCAL ***records***, and here is the definition of their structured type:

```
type
   oprange = 1..30  ;
   NODEP   = ^ NODE ;             (* node pointer                     *)
   NODE    = record
      ntype      : (copy, copytranslated, inport, other) ;
      nlabel     : integer ;
      ntext      : alfa    ;
      nnoofrefs  : integer ;      (* number of references (COPY) *)
      nnoofops   : 0..30   ;      (* number of operands           *)
      nop        : array [oprange] of NODEP   ;
      ninitop    : array [oprange] of integer ;
   end ;
```

Explanation: among the fields of every NODE record, the following are readily preset in the course of the Lucid graph definition:

ntype — set to copy if the node is a COPY node (and it is further changed to copytranslated in the course of translation), it is set to inport if the node stands for an inport, and it is otherwise set to other.

ntext — preset with a string fully specifying the node type,

nnoofrefs — preset with the number of references (1, 2, ...).

nnoofops — preset with the number of operands (0, 1, 2, ...).

nop — preset with pointers to the operand nodes

Every UDF inport is expressed through a NODE record whose ntype is inport, with the inport number (1, 2, ...) stored in the nlabel field. The fields nlabel and ninitop convey node numbers and are essential for the translation.

The Translation Program

We will now describe the recursive function Translate, together with a few assisting routines, which performs the translation. Translate must be applied to one program segment (one subnet) after another. At every translation step we have a particular *N*ode *U*nder *C*onsideration, we call it the "**NUC**". At the beginning of the translation of any program segment we choose the highest ranking node as the NUC.

We attach a label number to each node, and we achieve this by a function NextLabel which delivers successive integers. Our algorithm will ensure that the highest ranking node gets the "0" label; inferiors get label numbers higher than their superiors.

```
function NextLabel (var nodenumber : integer) : integer ;
   begin NextLabel  := nodenumber     ;   (* pseudo function *)
         nodenumber := nodenumber + 1 ;
   end ;
```

The procedure ScanOperands inspects left to right all the operands of the NUC. It translates each operand appropriately, by recursion to Translate, and it encodes in the ninitop field of NUC how each operand will eventually be retrieved in the initialisation of NUC. Inport nodes do not map into actors; they get therefore separate treatment which does not involve Translate.

```
procedure ScanOperands (nuc : NODEP; var nodenumber : integer) ;
   var  i      : integer ;
        nucop  : NODEP   ;
   begin with nuc^ do
     for i := 1 to nnoofops
     do  begin
         nucop := nop[i] ;
         if    nucop^.ntype = otinport
         then ninitop[i]  := -nucop^.nlabel          (* inport *)
         else ninitop[i]  := Translate(nucop, nodenumber) ;
   end   end ;
```

The procedure NodeInitialisation translates the information from the the ninitop field of NUC into the actual instruction for the actor initialisation. Use of the ninitop field is difficult to avoid. For any node actor, all operands must be created and initialised before initialisation of the actor itself. A COPY node actor must deliver its own name and also the names of all its outport actors (the references to the COPY) *before* it can be initialised itself. In the translation of any particular node, ScanOperands is always called in the *first* invocation of Translate, while NodeInitialisation is called in the *last*. This first and last invocation are the same for most node actors, only COPY node actors have more than one reference.

```
procedure NodeInitialisation (nuc : NODEP) ;
   var  i : integer ;
   begin with nuc^ do begin
     write ('    SEND (DATON, ') ;

     for i := 1 to nnoofops
     do  begin
         write ('node[',  ninitop[i]) ;
         if   i < nnoofops then  write ('], ') ;
         end ;

     writeln ('])  TO (node[', nlabel, ']) ;')  ;
   end end ;
```

The function $\boxed{\text{Translate}}$ takes a NUC pointer, and generates the whole ***creation*** and ***initialisation*** code for the corresponding actor. It generates that code also for all node operands. The result of function $\boxed{\text{Translate}}$ is the label (subscript *i* in $\boxed{\text{node[i]}}$) of the actor which takes the place of the NUC. Note the split actor labelling in the case of $\boxed{\text{COPY}}$ nodes. $\boxed{\text{COPY}}$ nodes constitute probably the most challenging part of the translation, and the algorithm contains some extra treatment for the benefit of $\boxed{\text{COPY}}$ nodes. The stages of the translation are always:

a) allocate a label for the new actor

b) ($\boxed{\text{COPY}}$: allocate one more label for the inport actor,)

c) generate a $\boxed{\text{CREATE}}$ for the actor,

d) translate the operands,

e) ($\boxed{\text{COPY}}$: generate an "obtain name of outport actor",)

f) if this has been the last reference, generate the initialisation,

g) return with label of the NUC.

Stages b) to d) are omitted if the NUC is a $\boxed{\text{COPY}}$ which has been touched before. $\boxed{\text{Translate}}$ is a ***pseudo*** function since it changes its operands. Here now is the all–important function $\boxed{\text{Translate}}$ (the program in its entirety is listed in appendix C):

```
function Translate (nuc : NODEP; var nodenumber : integer ) : integer ;
   var
      transl    : integer ;          (* new node will be node[(transl)] *)
   begin
   with nuc^ do begin
      transl    := NextLabel(nodenumber) ;
      translate := transl ;          (* the function result             *)

      if   ntype <> copytranslated
      then begin
           if   ntype = copy
           then begin  ntype   := copytranslated ;
                       nlabel  := NextLabel(nodenumber) ;
                end
           else        nlabel  := transl ;

           writeln ('   node[', nlabel,
                    '] := CREATE(Act_', ntext, ') ;') ;

           ScanOperands (nuc, nodenumber) ;
           end ;

      if   ntype = copytranslated
      then writeln ('   ( , , node[', transl,
                    ']) := RECEIVE FROM (node[', nlabel, ']) ;') ;

      nnoofrefs := nnoofrefs - 1 ;
      if   (nnoofrefs = 0)  and  (0 < nnoofops)
      then NodeInitialisation (nuc);

   end end ;
```

4.3.5 Concluding Remarks about the Lucid Graph Translation

In the presentation of the ***universal node act*** (↑ 4.1) we have subdivided the LUX code into two parts:

Y–part which is executed each time a request is sent to the operator actor in question, and

X–part which is executed once before the first execution of (Y).

A second glance at Act_Root and either of the UDF acts might tempt us to ***generalise*** that the Y–part is of considerable size and varies greatly from one program to another, while the X–part is at best small and of little variation. However, such a generalisation is true only for code from the translation algorithm described so far.

Various code refinement techniques will be presented in chapter VI, and that observation will no longer be valid.

The LUX code from the above translation ($\boxed{\text{Act\_Root}}$ and UDF acts) has its strong and its weak sides. Its merits lie in its ease of production, and in its accessibility to various analyses. We will carry out such analyses in chapter VI. The code is comprehensible but leaves wishes for elegance unfulfilled. This could be overcome by a table-driven universal ***subnet creation procedure***.

Although the code allows a bearably efficient implementation of concurrency, its ***efficiency*** leaves wishes open. Since we are using a demand driven evaluation strategy, most of the actors will be dormant for most of the time. In most implementations, the cost per actor is relatively high. Actors should be reserved for situations where concurrency is of true benefit, and they should not be kept around in dormant state. In chapter VI, we will look at ways of improving the efficiency of certain parts of the code much further, and in particular how to restrict concurrency to productive roles.

4.4 Memory in Node Actors

We know that, in demand driven DF, datons are evaluated only upon an explicit request. This means, whenever a daton appears somewhere, there must have been a preceding request for its evaluation. We can even state precisely ***where*** the daton queues build up:

> Theorem: In demand driven DF, daton ***queues*** need to be permitted only at the outports of COPY nodes.

This is a strong claim, but it is easy to prove. A long-term daton queue will certainly not build up at an ***inport*** of a node, since once a node (superior) issues a daton request to another node (inferior), the superior will consume the daton as soon as the inferior can deliver. For the same reason, a long-term daton queue will not build up at the ***outport*** of a node with only ***one*** outport. The node (with the one outport) will have produced the daton only in response to a request, and the superior will consume that daton as soon as it becomes available. Matters are rather different at the outport of a COPY node. Every COPY node links a number of outports to one inport, and a request on a single outport is enough to cause a request at the COPY inport. Therefore, if a daton arrives at the COPY inport, the COPY node will pass it on to the requesting outport(s), but it will have to ***queue*** it at all other outports.

In this thesis, FBY, NEXT, IF and the usual pointwise operators are called ***primitive operators*** (↑ 4.3.3.1). Their acts can be designed so that none of them has ***long***—term memory. Each of their actors is in exactly the same state whenever it is dormant; their storage cells hold only ***short***—term information (except for operand names, which are ***quasi***—constants anyway), nor does the PC hold state information. ***Previous*** requests have no lasting effect on primitive node actors. Optimisation can take advantage of this property (act expansion, ↑ 6.2). — On the other hand, FIRST, UPON and WVR certainly ***have*** memory, and *UDFs* are clearly entitled to having memory. We will indeed implement FIRST, UPON, and WVR (they all have memory) through UDFs.

4.5 Node Acts

The design of the node acts is presented only as late as now since this order of presentation appears to be the most natural one: the ***underlying concept*** has been explained at length, so that the focus can now be shifted to ***technical*** points. Some readers may by now have an inkling what the acts must look like.

The complexity varies considerably among the node acts. The more inports and outports a node has, the more protocol states its act must keep in harmony. We intend to exploit the request protocol to the full, and this makes the node acts rather complex. Some of the simpler acts have already been explained earlier on, the more difficult ones will be dealt with in the following. Simplest–to–hardest they are:

- any node which has only an inport or only an outport (e.g. constant, READ, and WRITE, ↑ 4.5.4 and 4.5.9 f),
- any node with one outport and one inport (↑ 3.4.4),
- any node with one outport and *more* than one inport, with ***sequential*** acquisition of its operands (↑ 4.5.2),
- any node with one outport and more than one inport, with ***concurrent*** acquisition of its operands (↑ 4.5.3),
- important special nodes (IF, FBY, NEXT, ↑ 4.5.6 ff),
- COPY nodes (↑ 4.6).

Each node act must be able to handle the *full* request protocol (i.e. COMPUTE, NULLIFY, ADVANCE). There would be no gain in clarity if we studied nodes which can handle only a simplified protocol. Appendix D gives some examples of OCCAM equivalents.

This section will not present acts for FIRST, UPON, WVR, or ASA. Our translation does not treat these operators as fundamental operators but as UDFs (↑ 5.6 and 6.6). Their function definitions are:

```
Wvr (a, k)   =  IF   First (k)  THEN  p  ELSE  q  FI
                WHERE p = a FBY q ;
                      q = Wvr (NEXT a, NEXT k) ;
                END ;

Upon (a, k)  =  a FBY Upon (p,  NEXT  k)
                WHERE p = IF   First (k)
                          THEN (NEXT  a)
                          ELSE        a    FI ;
                END ;

First (a)    =  p  WHERE   p = a FBY p   END ;

Asa (a, k)   =  First (Wvr (a, k)) ;
```

A simple-minded UDF implementation of these functions would be extremely wasteful, in particular in the case of $\boxed{\text{WVR}}$, but these UDFs can be optimised into perfectly efficient code (↑ 6.6).

4.5.1 Function $\boxed{\text{GetDaton}}$

The explanation of one other thing seems in place before we delve into node acts. The LUX function $\boxed{\text{GetDaton}}$ has been presented in section 3.4.3 as illustration for some aspect of LUX syntax. But that function is of more than mere syntactic interest; it is actually ***used*** in almost every node act. It deserves therefore more than mere passing mention. We will now explain it formally, but its full importance will become evident when we study its applications in the subsequent sections. Here is the function again:

```
FUNCTION GetDaton (index : INTEGER ; operand : ACTOR) : ANYTYPE ;
   LABEL 1 ;
   BEGIN                                                        :R
     SEND (COMPUTE, index) TO (operand) ;                       :1
     ( , , GetDaton) := RECEIVE FROM (operand) ;
     RETURN ;    (* normal RETURN even if exception occurred. *)

1:   EXCEPTION (NULLIFY, index) TO (operand) ;                  :R
   END ;
```

$\boxed{\text{GetDaton}}$ sends a $\boxed{\text{COMPUTE}}$ request to the operand actor, and awaits then the arrival of a the requested daton value. That daton value is eventually returned as the function result. A typical application would be:

```
                                                  :7
onedaton := GetDaton (thisindex, op_node_actor) ;
```

This LUX instruction requests from $\boxed{\text{op\_node\_actor}}$ that the daton at $\boxed{\text{thisindex}}$ be evaluated, and once that has been achieved the daton value is stored in $\boxed{\text{onedaton}}$. If an exception occurs, the outcome depends on how far we got in the function execution:

- If the exception occurs ***before*** the operand daton has been requested (SEND COMPUTE ...), a special return is made right away, namely through the door (.7) immediately ***before*** the function call. (Program execution continues at label 7., not shown in the example).
- No special action is taken if the exception occurs ***after*** the operand daton has been received. Instead, normal execution continues and an ***ordinary*** return is made (i.e. no door is used). This gives us a chance to ***preserve the daton value***. This course of action is appropriate: the purpose of NULLIFY exceptions was the abortion of over–long computations, but after the receipt of the result daton this purpose has lost its urgency.
- If, however, the exception occurs ***after*** the COMPUTE request but ***before*** the arrival of the daton, a NULLIFY exception is sent to op_node_actor, followed by a special return using the door (.7) before the function call. The NULLIFY exception nullifies the daton evaluation in the inferior

The node acts and the request protocol have been designed under the guideline that, once a node actor has received an exception, it must not carry out any further computation, except for some concluding administration. In general, it is hard to tell which intermediary result is so valuable as to deserve preservation (there is scope for an optimiser).

4.5.2 Acts which Request their Operands Sequentially

When implementing the operators of a programming language, one is tempted to contemplate two kinds of ***variants*** of each operator:

(a) variants which make better use of the computer resources (faster execution or lower store requirements).

(b) variants which maximise the output history of the program (some operators cause subnets to produce shorter output histories than one might expect).

This thesis is not much concerned with category (a) of variants ("***local optimisation***"). For example, once it has been specified that a von Neumann monoprocessor is the computer to be used, there is hardly any scope left for improvements in category (a). Section 4.5.3 will show that some progress can be made in category (b). For example, once either operand of $\boxed{\text{OR}}$ yields $\boxed{\text{TRUE}}$, the other operand's daton value is no longer required. This can be exploited by ***concurrent*** operand evaluation. (Pseudo–) Concurrency is rather costly on von Neumann monoprocessors, and should be reserved for special cases.

For most operators, such refinement is impossible anyway. Most operators cannot dispense with any of their operand datons; ***sequential*** operand evaluation (i.e. one operation after another) is therefore the appropriate method when dealing with von Neumann monoprocessors.

Example ($\boxed{\text{Act\_Plus\_}}$)

The following act implements an operator which acquires its operands in sequential order. The example describes the binary "+" operator, but all those ***pointwise*** operators which unconditionally need ***all*** their operands (e.g. relationals) have very similar acts.

```
ACT Act_Plus_ ;                                          (* Sequential PLUS *)
    LABEL 1 ;
    VAR
       dval0, dval1, result : REAL  ; (* or whatever the daton type *)
       superior, p0, p1     : ACTOR  ;
       request              : MSGTYPE ;
       index                : INTEGER ;
    BEGIN                       (* p0 and p1 are the operand actors.     *)
      ( , , p0, p1) := RECEIVE FROM (Creator) ;

(* ——— building block 1: start loop ——————————————— *)
    REPEAT
      WHILE TRUE DO
      BEGIN                                                              :1
        (superior, request, index) := RECEIVE () ;

(* ——— building block 2: get operand ——————————————— *)

                      (* Possibly hang up in GetDaton.            *) :1
          dval0 := GetDaton (index, p0) ;   (* Get 1st operand.     *)

(* ——— building block 2: get operand ——————————————— *)
                                                                         :1
          dval1 := GetDaton (index, p1) ;   (* Get 2nd operand.     *)

(* ——— building block 3: send the result and end loop ——————— *)
                                                                         :1
        result := dval0 + dval1 ;           (* node dependent        *)
                      (* Possibly hang up in SEND.                *) :1
        SEND (DATON, result) TO (superior) ;
      END ;           (* End of inner eternal loop.               *)

1:    (request, index) := Reveal ;          (* Exception part.      *)
      IF    request = ADVANCE
      THEN  EXCEPTION (request, index) TO (p0, p1) ,
      RESET ;
    UNTIL FALSE ;     (* End of outer eternal loop.               *)
    END ;             (* End of Act_Plus_ .                       *)
```

In the initialisation, the node actor learns who its operand actors are. After that, the node actor enters an eternal loop in which it successively processes requests. The act is easier to understand if we pretend first that there are *no* exceptions: we can ignore all doors and the exception part. Upon arrival of a COMPUTE request, the operand datons are acquired one after the other, the result value is computed, and the result is then sent back to the superior. After that, the node actor is ready to accept the next request.

However, a NULLIFY or an ADVANCE exception *can* occur anywhere within that loop. If this happens while an operand daton is under way (requested but not yet obtained), the inferior computation is aborted by giving a NULLIFY exception to the operand actor (↑ explanation in 4.5.1). As soon as we reach a door we break out of the usual order of instruction execution and proceed with the *exception* part. If the exception was an ADVANCE, the ADVANCE is *propagated* to all the operand actors. The exception handling ends with executing RESET. After that, the node actor is ready to accept the next request.

Acts for deterministic (i.e. avoidably concurrent) pointwise operators with *other* than two operands can be built up from the building blocks of Act_Plus. In particular block 2, acquisition of an operand, can be reduplicated for the acquisition of any number of operands. The beginnings and endings of the act propers are practically identical. (Exercise for the reader: write the act for a constant, solution in 4.5.9.)

Clearly every instruction has, where possible, been furnished with an "escape route" (viz. a *door*) for the event of an exception. The computation proper result := ... has, in our example, been very simple and inexpensive, its escape route was therefore dispensable.

4.5.3 Acts which Request their Operands Concurrently

We sketched above (also ↑ 1.1.3) the benefits of certain concurrent computations. In computers where concurrency is cheap (e.g. *transputers*) it would even be *advisable* to implement most operators with as much concurrency as possible. We study in this section how to design node acts which acquire their operand datons concurrently.

In mathematics, the *sequencing* of the operands has no bearing on the result of a commutative operator, by definition. Implementations of many programming languages, however, treat operators like OR and AND as non-commutative. One of the aims of Lucid is to bring mathematics and programming closer together. Concurrency can help us in this pursuit (↑ 1.1.3).

Example (Act_Or_)

The following act implements a binary operator using ***concurrent*** acquisition of the operand datons. The example represents the OR operator, but every other binary pointwise operator whose result may be determined by the daton arriving first (e.g. AND, multiply with ***zero test***) would have a very similar act.

```
ACT Act_Or_ ;                                          (* Concurrent OR *)
    LABEL 1 ;
    VAR
       superior, p0, p1, other, sender : ACTOR ;
       request : MSGTYPE ;
       index   : INTEGER ;
       dvalue  : BOOLEAN ;
    BEGIN                      (* p0 and p1 are the operand actors.    *)
       ( , , p0, p1) := RECEIVE FROM (Creator) ,

    REPEAT
      WHILE TRUE DO
      BEGIN                                                             :1
        (superior, request, index) := RECEIVE () ,                      :1

        SEND  (COMPUTE, index)         TO   (p0, p1) ;                  :1
        (sender, , dvalue) := RECEIVE FROM (p0, p1) ;                   :1
        IF   sender = p0  THEN  other := p1  ELSE  other := p0 ;        :1

        IF   dvalue     (* Inspect what has been obtained so far.  *)  :1
          (* ^^^^^^ node dependent                                  *)
        THEN EXCEPTION (NULLIFY, index)     TO   (other)
        ELSE ( , , dvalue) := RECEIVE FROM (other) ;
                                                                        :1
        SEND (DATON, dvalue) TO (superior) ;
      END ;

1:    (request, index) := Reveal ;          (* Exception part.         *)
      EXCEPTION (request, index) TO (p0, p1) ,
      RESET ;
    UNTIL FALSE ;
    END ;                   (* End of Act_Or_ .                        *)
```

For the AND and super-multiply act, practically all lines involving dvalue must be reformulated, of course, but the overall structure will remain unchanged.

There are considerable commonalities between the $\boxed{\text{Act\_Or\_}}$ and the $\boxed{\text{Act\_Plus\_}}$; the differences lie in the code which deals with the operand acquisition. The $\boxed{\text{GetDaton}}$ function ***cannot*** be used here since it has been tailored for acquiring datons sequentially.

The initialisation (unchanged) is still followed by an eternal loop in which the node actor successively processes all requests. First, let us again pretend that there are no exceptions. Upon arrival of a $\boxed{\text{COMPUTE}}$ request, that request is propagated to ***both*** operand actors at the same time. After that, a reply is awaited from ***either*** operand actor. (A ***random*** pick is taken if both replies become available at the same moment.) Once the first reply has been received, the ***remaining*** operand actor is due to be dealt with; a quick test works out its actor name $\boxed{\text{other := ...}}$. The value of the ***first*** reply decides over the next action. The $\boxed{\text{other}}$ operand is sent a $\boxed{\text{NULLIFY}}$ iff its daton value is now ***irrelevant*** (that $\boxed{\text{NULLIFY}}$ is the same no matter whether that daton's evaluation is ***complete***, or whether it is still under way). ***Otherwise***, the completion of the $\boxed{\text{other}}$ operand evaluation is awaited. Either way, once both operand actors are dormant again, the overall result value is ***worked out*** and is sent back to the superior. After that, the node actor is ready to accept the next request.

A $\boxed{\text{NULLIFY}}$ or an $\boxed{\text{ADVANCE}}$ exception ***can*** occur anywhere within that loop. If this happens while any operand daton is still under way (requested but not yet obtained), any inferior computation must be aborted by sending $\boxed{\text{NULLIFY}}$ exceptions to the operand actors. Whenever an exception occurs, it is propagated unchanged to both operand actors. The exception handling ends with executing $\boxed{\text{RESET}}$. After that, the node actor is ready to accept the next request.

$\boxed{\text{Act\_Or\_}}$ uses a somewhat crude method of exception propagation ($\boxed{\text{NULLIFY}}$ requests are propagated ***unconditionally***), but this degrades the efficiency of program execution only very little. Luckily, sending a $\boxed{\text{NULLIFY}}$ to a dormant node actor causes only negligible extra work. It is easy to extend the code of $\boxed{\text{Act\_Or\_}}$, making it propagate $\boxed{\text{NULLIFY}}$ only to those operand actors which are ***busy*** with work.

Generating a $\boxed{\text{NULLIFY}}$

This section has introduced one new concept, namely ***nullifying*** a computation after it has been set in motion. It must be born in mind that this mechanism can be used simultaneously on ***numerous*** levels. Take, for example, a Lucid expression with an $\boxed{\text{OR}}$ in it, of which either operand is a subexpression with a further $\boxed{\text{OR}}$ (or $\boxed{\text{AND}}$) in it. Such nestings can be constructed to any depth. During the evaluation of such an expression, any $\boxed{\text{OR}}$ node actor may decide to nullify the evaluation of its operands. This will nullify all inferior evaluations.

4.5.4 The $\boxed{\text{WRITE}}$ Act

As far as act construction is concerned, we have learnt how to build UDF acts and how to build the acts for the simpler operators. In both cases the end product could be built by applying a few simple rules to a few standard building blocks. We will now have a look at individual acts, and in particular at acts which do not fit readily into the general pattern. $\boxed{\text{WRITE}}$ and $\boxed{\text{READ}}$, the Lucid specific operators $\boxed{\text{FBY}}$ and $\boxed{\text{NEXT}}$, and last not least $\boxed{\text{COPY}}$ are among them.

A program without any $\boxed{\text{WRITE}}$ node would be pointless. In demand driven evaluation, the ***driving force*** for all computations stems ultimately from a $\boxed{\text{WRITE}}$ node. Here is the $\boxed{\text{WRITE}}$ act:

```
ACT Act_Write_ (filename : ALFA) ;
    VAR
       index : INTEGER ;
       p0    : ACTOR   ;
    BEGIN
        ( , , p0) := RECEIVE FROM (Creator)    ;
        index     := 0 ;
        OPEN (filename, WRITEmode) ;
                     (* WRITEmode is a system constant. *)
      REPEAT
        WRITE (filename,   GetDaton (index, p0) ) ;
        index  := index + 1  ;
        EXCEPTION (ADVANCE, index)      TO (p0)    ;
      UNTIL FALSE ;  (* End of eternal loop.          *)

    END ;
```

The $\boxed{\text{Act\_Write\_}}$ does not receive any requests, and needs therefore no exception handling. – During program execution masses of requests (including exceptions) pulsate through the net of node actors; it is interesting to note that the origin of most $\boxed{\text{COMPUTE}}$ and $\boxed{\text{ADVANCE}}$ requests can be traced back to $\boxed{\text{Act\_Write\_}}$. – The special role of $\boxed{\text{WRITE}}$ actors has repercussions on their scheduling priority (↑ 4.7).

4.5.5 The Daton Sink Act

```
ACT Act_Daton_Sink_ ;
    VAR
       p0 : ACTOR ;
    BEGIN
        ( , , p0) := RECEIVE FROM (Creator) ;
        EXCEPTION (ADVANCE, finalindex) TO (p0)      ;
        (* This act needs no eternal loop.      *)
    END ;
```

The act of the ***daton sink node*** is presented here for dramatic relief. This node is the poor relative of the $\boxed{\text{WRITE}}$ node, all comments about exceptions and scheduling apply correspondingly. Its effect is like writing to a ***null*** device, and its only foreseeable application is with multi-valued UDFs, although such UDFs can not be expressed in present Lucid.

The $\boxed{\text{Act\_Daton\_Sink}}$ generates only one request ever, namely the special request $\boxed{\text{ADVANCE, finalIndex}}$ ($\boxed{\text{finalIndex}}$ is a special constant, not a natural number). This request states that there will be no requests for further datons ever. Considering that we are dealing with a demand driven evaluation scheme, this is the ***ultimate non–demand***. More on this in section 4.5.6.

4.5.6 The $\boxed{\text{FBY}}$ Act

$\boxed{\text{PLUS}}$ and $\boxed{\text{OR}}$ are both pointwise nodes (consequence: whenever, say, $\boxed{\text{Act\_Plus}}$ propagates a request to one of its operand actors, this request goes ***with its index unchanged*** from the original request; the request index is described in ↑ 4.2). Neither $\boxed{\text{FBY}}$ nor $\boxed{\text{NEXT}}$ is pointwise; their acts propagate a ***modified*** request index. This makes their acts only slightly more complicated. At certain index values some special action is required, most of it in the exception handling. Here is the act for the $\boxed{\text{FBY}}$ node:

```
ACT Act_Fby_ ;
    LABEL 1 ;
    VAR
      superior, p0, p1 : ACTOR ;
      request : MSGTYPE ; index : INTEGER ; result : ANYTYPE ;
    BEGIN
      ( , , p0, p1) := RECEIVE FROM (Creator) ;

    REPEAT
      WHILE TRUE DO
      BEGIN                                                    :1
        (superior, request, index) := RECEIVE () ;

        IF   index = 0
        THEN                                                   :1
             result := GetDaton (index, p0)
        ELSE                                                   :1
             result := GetDaton (index-1, p1) ;
                                                               :1
        SEND (DATON, result) TO (superior) ;
      END ;               (* End of inner eternal loop.      *)

1:    (request, index) := Reveal ;
      IF   request = ADVANCE
      THEN BEGIN
             IF        index = 1
             THEN      EXCEPTION (request, finalindex) TO (p0)
             ELSE IF   index = finalindex
                  THEN EXCEPTION (request, index     ) TO (p0, p1)
                  ELSE EXCEPTION (request, index-1   ) TO     (p1) ;
           END ;
      RESET ;
    UNTIL FALSE ;         (* End of outer eternal loop.       *)
    END ;                 (* End of Act_Fby_ .                *)
```

The FBY node has one peculiarity, and this is reflected in the FBY act. At best, just ***one*** daton (viz. the initial daton) is acquired from operand actor p0. After that, the operand actor for p0 is notified that no further daton will ***ever*** be requested. This is expressed by the request ADVANCE, finalindex. Without the latter request, immense ***queues*** might build up inside any COPY node involved in the evaluation of operand p0. The reason is easy to see. Assume the ADVANCE, finalindex request did not exist, and consider a COPY node which has not received any request on outport X for a long time, while at the same time outport Y has delivered many datons. The COPY would not be able to decide whether outport X has actually ***died***, it will never request again. Instead the COPY would have to stay ready (and retain all the daton values) for an eventual COMPUTE request on outport *X*.

The special ADVANCE request solves this problem by providing extra information. As a penalty, the exception handling becomes more difficult. Instead of the ADVANCE request with a special index value we could have added a new request type LAST with the same effect, although that would have increased the code of *all* node acts. More on this topic in the discussion of the KILL request (↑ 6.3).

4.5.7 The NEXT Act

```
ACT Act_Next_ ;
    LABEL 1 ;
    VAR
      p0, superior : ACTOR   ;    request : MSGTYPE ;
      index        : INTEGER ;    result  : ANYTYPE ;
    BEGIN
      ( , , p0) := RECEIVE FROM (Creator) ;

      EXCEPTION (ADVANCE, 1) TO (p0) ;

    REPEAT
      WHILE TRUE DO
      BEGIN                                                  .1
        (superior, request, index) := RECEIVE () ;
                                                             :1
        result := GetDaton (index+1, p0) ;                   .1
        SEND (DATON, result) TO (superior) ;
      END ;             (* End of inner eternal loop.      *)

1:    (request, index) := Reveal ;
      IF          request = ADVANCE
      THEN BEGIN
             IF   index   = finalIndex
             THEN EXCEPTION (request,     index) TO (p0)
             ELSE EXCEPTION (request, 1 + index) TO (p0) ;
           END ;
      RESET ;
    UNTIL FALSE ;       (* End of outer eternal loop.      *)
    END ;               (* End of Act_Next_ .              *)
```

The NEXT node actor issues one bare ADVANCE request before propagating its initial request. (Any *bare* ADVANCE originates from NEXT or from IF.) Moreover, the index is increased by one in all propagated requests. In all other respects, NEXT resembles closely a pass-through node.

eventual COMPUTE request on outport *X*.

The special ADVANCE request solves this problem by providing extra information. As a penalty, the exception handling becomes more difficult. Instead of the ADVANCE request with a special index value we could have added a new request type LAST with the same effect, although that would have increased the code of *all* node acts. More on this topic in the discussion of the KILL request (↑ 6.3).

4.5.7 The NEXT Act

```
ACT Act_Next_ ;
    LABEL 1 ;
    VAR
      p0, superior : ACTOR    ;     request : MSGTYPE ;
      index        : INTEGER ;     result  : ANYTYPE ,
    BEGIN
      ( , , p0) := RECEIVE FROM (Creator) ;

      EXCEPTION (ADVANCE, 1) TO (p0) ;

    REPEAT
      WHILE TRUE DO
      BEGIN                                                    :1
        (superior, request, index) := RECEIVE () ;
                                                               :1
        result := GetDaton (index+1, p0) ;                     :1
        SEND (DATON, result) TO (superior) ;
      END ;              (* End of inner eternal loop.        *)

1:    (request, index) := Reveal ;
      IF           request = ADVANCE
      THEN BEGIN
             IF   index   = finalIndex
             THEN EXCEPTION (request,     index) TO (p0)
             ELSE EXCEPTION (request, 1 + index) TO (p0) ;
           END ,
      RESET ;
    UNTIL FALSE ;        (* End of outer eternal loop.        *)
    END ,                (* End of Act_Next_ .                *)
```

The NEXT node actor issues one bare ADVANCE request before propagating its initial request. (Any *bare* ADVANCE originates from NEXT or from IF.) Moreover, the index is increased by one in all propagated requests. In all other respects, NEXT resembles closely a pass-through node.

We know that the fundamental acts other than COPY have no memory. This is little surprise in the case of pointwise operators like IF. However, one would expect that FBY and NEXT differentiate at least between an *initial* state and a *continuation* state. However, Act_Next_ progresses right after initialisation to its continuation state, whereas Act_Fby_ deduces the state from the index in the request.

The daton index changes only in the course of ADVANCE requests, and each ADVANCE comes normally with its index one greater than the previous index. ADVANCE, finalindex is the only exception to this rule. Only WRITE, READ and COPY node actors need to remember which daton is next in line.

4.5.8 The IF Act

```
ACT Act_Ite_ ;                                          (* IF-THEN-ELSE *)
    LABEL 1 ;
    VAR
      superior, p0, p1, p2      : ACTOR    ; request  : MSGTYPE ;
      index : INTEGER ; condi : BOOLEAN ; result   : ANYTYPE ;
    BEGIN
      ( , , p0, p1, p2) := RECEIVE FROM (Creator) ;

    REPEAT
      WHILE TRUE DO
      BEGIN                                                            :1
        (superior, request, index) := RECEIVE () ;                     :1
        condi := GetDaton (index, p0) ;

        IF   condi
        THEN (* EXCEPTION (ADVANCE, index+1) TO (p2)            *) :1
             result := GetDaton (   index,        p1)
        ELSE (* EXCEPTION (ADVANCE, index+1) TO (p1)            *) :1
             result := GetDaton (   index,        p2) ;
                                                                       :1
        SEND (DATON, result) TO (superior) ;
      END ;              (* End of inner eternal loop.          *)

1:    (request, index) := Reveal ;
      IF    request = ADVANCE
      THEN  EXCEPTION (request, index) TO (p0, p1, p2) ;
      RESET ;
    UNTIL FALSE ;        (* End of outer eternal loop.          *)
    END ;                (* End of Act_Ite_ .                   *)
```

In the eternal loop, the IF node actor interrogates first the operand p0, the IF condition. Dependent on the value of that daton, either the THEN operand p1 or the ELSE operand p2 is selected to constitute the overall result. — This Act_Ite_ contains nothing which exceeds the general construction pattern from section 4.5.2; chapter VI will give hints how to refine IF (constant condition and concurrent IF). A possible refinement has been sketched: the ADVANCE exception can be issued to the ***rejected*** operand at a very early time. However, to implement this properly requires some adjustments: either successive ADVANCE requests with the same index must be permitted, or the condi value must be retained in ***memory***.

4.5.9 The Constant Act

Every program must get ***data*** from somewhere, be it data read from a ***file***, or ***constants***. READ and the constants are the two fundamental nodes which have only an ***outport***. Obviously, the act of neither needs initialisation. Here is the act for a constant delivering node:

```
ACT Act_Const_ (consta : ANYTYPE) ;
    LABEL 1 ;
    VAR
      superior : ACTOR ; request : MSGTYPE ; index : INTEGER ;
    BEGIN            (* act has no initialisation.       *)

      REPEAT
        WHILE TRUE DO
        BEGIN                                              :1
          (superior, request, index) := RECEIVE () ;       :1
          SEND (DATON, consta) TO (superior) ,
        END ,           (* End of inner eternal loop.     *)

1:      RESET ;
      UNTIL FALSE ;     (* End of outer eternal loop.     *)
    END ;
```

Each Act_Constant_ actor gets a kind of initialisation during its own creation: the value of the constant itself. There is nothing else to explain in this act. The READ act is similar, except that everything is much more complicated:

4.5.10 The READ Act

```
ACT Act_Read_ (filename : ALFA) ;
    LABEL 1 ;
    VAR
      superior : ACTOR   ; index  : INTEGER ;
      request  : MSGTYPE ; index2 : INTEGER ; result : ANYTYPE ;
    BEGIN     (* There is no initialisation message from the creator. *)
        OPEN (filename, READmode) ; (* READmode is a system constant. *)
        index2 := 0 ;

    REPEAT
      WHILE TRUE DO
      BEGIN                                                          :1
        (superior, request, index) := RECEIVE () ;                   :1

        (* IF   index <> index2  THEN ReportError ; *)
                                                                     :1
        result := READ (filename, index2) ;                          :1
        SEND (DATON, result) TO (superior) ;
      END ;               (* End of inner eternal loop. *)

1:    (request, index) := Reveal ;
      IF   request = ADVANCE           (* this test can be omitted. *)
      THEN BEGIN
             IF   index = finalindex
             THEN CLOSE (filename)
             ELSE index2 := index2 + 1 ,
               (* IF  index <> index2  THEN ReportError ;          *)
           END ,
      RESET ;
    UNTIL FALSE ;         (* End of outer eternal loop.              *)
    END ;                 (* End of Act_Read_ .                      *)
```

The index2 in the instruction READ (..., index2) refers to the running index of the daton in the file. This makes it possible to deliver the *same* daton upon successive COMPUTE requests of ***identical index***, as required by the protocol. In any implementation, Act_Read_ is likely to have memory of some form (viz character buffers etc.), but this memory contains only quasi–constants.

Every request quotes a particular index. The index can only be advanced by ADVANCE requests, and every ordinary ADVANCE request brings an increment of one. The READ node (and similarly COPY) needs the index information only to identify the special ADVANCE, finalindex requests. The index information can, however, be used to supervise the correct functioning of the system, a running check like "***parity***". The total reliance on local counting ($\boxed{\text{index2}}$) creates an opportunity for optimisation (implicit $\boxed{\text{NEXT}}$ and $\boxed{\text{FBY}}$, ↑ 6.2).

Virtually all operating systems are ***data*** driven, and data are usually accessed sequentially, i.e. in a pipeline fashion. $\boxed{\text{WRITE}}$ and $\boxed{\text{READ}}$ actors interface to the operating system, and its characteristics shape, obviously, the design of the $\boxed{\text{WRITE}}$ and $\boxed{\text{READ}}$ acts. A ***demand*** driven $\boxed{\text{Act\_Read\_}}$ for reading interactively from terminals is a realistic proposition, and is quite easy to write. The $\boxed{\text{Act\_Write\_}}$ would also look quite different in a "tagged" DF operating system.

4.5.11 Exceptions in Primitive Acts

The description of the doors (↑ 3.4.2) may have appeared disproportionally complicated, considering their ***unsophisticated*** application in all the acts so far. Apparently, there was simply a door after almost every instruction, and the target was always the same. However, this looked so simple merely because all the difficult work has been shifted from the proper computing node actors to the $\boxed{\text{COPY}}$ node actors. In particular, most ***primitive*** nodes are without long–term memory. The exception handling of a primitive node is trivial:

1) it simply abandons its current work,

2) it propagates the exception to the operand actors (if appropriate),

3) it executes a $\boxed{\text{RESET}}$, and

4) it enters finally the dormant state.

This simple pattern would be totally inadequate for $\boxed{\text{COPY}}$, as we shall see. Even the action of UDFs (which can contain $\boxed{\text{COPY}}$ nodes) in the event of exceptions is much more complex; however, their exception action takes place within their ***internal*** node actors, and its complexity is therefore invisible.

4.6 The $\boxed{\text{COPY}}$ Act

4.6.0 Introduction

This section describes a $\boxed{\text{COPY}}$ act which imposes very few restrictions on its use. The only restriction is due to pipeline DF: datons must be requested in the order of *increasing* index. The maximum overall queue length (buffer size) is limited only by the *machine size*.

It is possible to implement each $\boxed{\text{COPY}}$ node as a ***single*** actor. However, such an actor would have to distinguish between a very large number of states, due to the many states each of its ports can be in (cross product). We choose a rather different approach, where each outport is implemented by its ***dedicated*** actor, with one further ***common*** actor for the inport. The $\boxed{\text{COPY}}$ inport actor is mainly concerned with the administration of the daton buffer. This design is ***modular***; each outport has only very little concern with the other outports. "**$\boxed{\text{COPY}}$ node actor**" is used meaning "all the actors which together implement the $\boxed{\text{COPY}}$ node".

The description of the $\boxed{\text{COPY}}$ node actor starts with general considerations, it explains then the outport act, and finally the inport act. The specific procedures are presented ***before*** each act.

4.6.1 Daton Buffers

The buffers are implemented as **chains** (= linked lists) with reference counts. In a $\boxed{\text{COPY}}$ with many outports, each outport buffer is organised as one linear linked list, with each outport "hooked in" at the appropriate place. The list store makes use of a pregiven ***store manager*** routine with ***explicit return*** of disused store space (the same store manager might also allocate all the actor space). The terms "***queue***", "***buffer***" and "***chain***" reflect merely different ***views*** of the same thing.

The following figure shows a daton chain of five buffer cells:

```
             (early)                                    (late)
in_index=t    t-5       t-4        t-3        t-2        t-1

             +---+     +---+      +---+      +---+      +---+
value        | I |     | L |      | R |      | O |      | Y |
             +---+     +---+      +---+      +---+      +---+
refcount     | 2 |     | 3 |      | 6 |      | 7 |      | 7 |
             +---+     +---+      +---+      +---+      +---+
linkage      |  -+---> |  -+--->  |  -+--->  |  -+--->  |NIL|
             +---+     +---+      +---+      +---+      +---+  NIL
                                                        tail
outports      ↑↑         ↑         ↑↑↑         ↑                 ↑
```

Every chain element can be referenced by ***any*** number of outports. The reference count states cumulatively the number of direct and indirect references. The uparrows in the bottom row symbolise those places where COPY outports are hooked into the chain. The arrow on the very right symbolises an outport referring to a ***future*** daton, while the buffer caters for ***past*** datons. That outport could be, for example, in the finalindex–state (i.e. referring to the "most distant" future daton). A COPY outport which, at a particular moment, refers to a queued daton can ***find the successor daton*** by following the link pointer. If the pointer is NIL, the next daton value needs to be ***evaluated*** beforehand and a new cell with that value ***appended*** to the chain. (The pointer value NIL means "***pointing nowhere***".) If required, the outport can eventually be ADVANCEd to the successor daton through "stepping forward" by means of the pointer, with the old reference count being decremented accordingly. The ***old*** cell can be ***released*** (given back to the store manager) once its reference count has dropped to zero.

We declare the buffer cells as follows:

```
TYPE
  CELLP   = ^ CELL   ;       (* Buffer cell pointer.          *)

  CELL    = RECORD           (* Store for one daton value.    *)
    value : ANYTYPE ;        (* The daton value.              *)
    count : INTEGER ;        (* Reference count.              *)
    link  : CELLP   ;        (* Pointer to next later cell.   *)
   END ;
```

The existence of a ***universal*** daton type is an illusion, of course; a string can hardly be stored in the same way as a Boolean. But implementors can find ways around that. For simplicity, we pretend from now on that all our data objects are of the hypothetical type ANYTYPE, and that they can all be held in storage cells of uniform size. We communicate with the store manager through two pregiven routines. Buffer cells are obtained by calling the parameterless function GetCell, and they are released by calling the procedure FreeCell. A simple minded program would go:

```
VAR mycell : CELLP ;
BEGIN
  mycell := GetCell ;
  ...
  FreeCell (mycell) ;
END ;
```

The chaining of the daton buffer cells brings considerable efficiency since with its aid the outports can share every buffer cell. This efficiency is sabotaged in a program with one COPY node feeding directly into ***another*** COPY node; such a construct should only be chosen in very select cases.

So far we have paid little attention to outports in **off-chain state**, i.e. outports which refer to ***future*** datons. Outports are put into that state by the receipt of numerous bare ADVANCE requests (* 4.2) or by ADVANCE, finalindex. Off-chain outports are not handled by the daton buffer but by a mechanism which will be described in section 4.6.4 (***request propagation***).

4.6.2 Protection by Semaphores

Whenever we access a stored data object, we trust in its consistency. (The data object may comprise many interrelated pieces of information.) For example, the number held in a reference counter is assumed to be equal, at any moment, to the ***factual*** number of references. Occasionally, however, data need to be changed, and inconsistent data may be unavoidable ***while*** the alteration is being carried out. The phase between the removal of a reference and the decrementing of the reference counter would be an example. The data should not be accessed "by the public" during such phases of inconsistency, and conversely, any interfering access must be locked out during phases of use. We need an "***access token***", where the holder of the token has the ***exclusive right of access*** to the data.

We use a **semaphore** to manage such an exclusive access right. It has been demonstrated in section 3.2.4 that semaphores can be implemented through message passing (Act_Guardian_). We will use that method here even though it may not be ideal in efficiency terms. The use of semaphores is easy. One semaphore is needed for each data object which needs protection at any moment. We create one semaphore by:

```
VAR semaphore : ACTOR ;
 ....
semaphore := CREATE (Act_Guardian_) ;
(* The semaphore is initially set to "access is public". *)
```

and, whenever necessary, we call:

```
MakeExclusive (semaphore) ;
 ...
MakePublic (semaphore) ;
```

where MakeExclusive and MakePublic are procedures which change the ***access status*** of the data object. While one actor upholds its claim to the data object (i.e. in the interval between MakeExclusive and MakePublic) any ***other*** actor calling MakeExclusive gets ***hung up*** until its turn has arrived. In our particular case both procedures are

4.6.2 Protection by Semaphores

Whenever we access a stored data object, we trust in its consistency. (The data object may comprise many interrelated pieces of information.) For example, the number held in a reference counter is assumed to be equal, at any moment, to the ***factual*** number of references. Occasionally, however, data need to be changed, and inconsistent data may be unavoidable ***while*** the alteration is being carried out. The phase between the removal of a reference and the decrementing of the reference counter would be an example. The data should not be accessed "by the public" during such phases of inconsistency, and conversely, any interfering access must be locked out during phases of use. We need an "***access token***", where the holder of the token has the ***exclusive right of access*** to the data.

We use a **semaphore** to manage such an exclusive access right. It has been demonstrated in section 3.2.4 that semaphores can be implemented through message passing (Act_Guardian_). We will use that method here even though it may not be ideal in efficiency terms. The use of semaphores is easy. One semaphore is needed for each data object which needs protection at any moment. We create one semaphore by:

```
VAR semaphore : ACTOR ;
  ...
semaphore := CREATE (Act_Guardian_) ;
(* The semaphore is initially set to "access is public". *)
```

and, whenever necessary, we call:

```
MakeExclusive (semaphore) ;
  ...
MakePublic (semaphore) ;
```

where MakeExclusive and MakePublic are procedures which change the ***access status*** of the data object. While one actor upholds its claim to the data object (i.e. in the interval between MakeExclusive and MakePublic) any ***other*** actor calling MakeExclusive gets ***hung up*** until its turn has arrived. In our particular case both procedures are

actually identical; they treat the semaphore like a *toggle* switch, and we have to be careful not to call either $\boxed{\text{MakeExclusive}}$ or $\boxed{\text{MakePublic}}$ twice in succession (one would use safer procedures in the real implementation). The procedures are:

```
PROCEDURE MakeExclusive (semaphore : ACTOR) ;
   BEGIN                                          :R
     SEND DATON TO (semaphore) ;  END ;
```

```
PROCEDURE MakePublic (semaphore : ACTOR) ;
   BEGIN                                          :R
     SEND DATON TO (semaphore) ;  END ;
```

4.6.3 Data Structures and Initialisation of $\boxed{\text{COPY}}$

A sizeable bank of information is accessed by the inport and each outport of $\boxed{\text{COPY}}$, and by procedures within them. Most of the state information of the inport and each outport can be grouped into data records (PASCAL's device for constructing data structures), which we call **descriptors**. This makes in particular the parameter passing much simpler. Here is the type declaration for our descriptors:

```
TYPE
  OUTPORTSTRUCT = RECORD          (* There is one of these per COPY outport.   *)
    oactor    : ACTOR   ;         (* Actor name of this outport.               *)
    buffer    : CELLP   ;         (* Daton buffer pointer.                     *)
    novalues  : INTEGER ;         (* How many leading datons shall be ignored *)
    oindex    : INTEGER ;         (* Current daton index (from last ADVANCE). *)
    waiting   : BOOLEAN ;         (* State indicator for inport.               *)
   END ;

  INPORTSTRUCT = RECORD           (* Inport descriptor:                        *)
    iactor    : ACTOR   ;         (* Actor name of this inport.                *)
    tailcell  : CELLP   ;         (* Pointer to tail of daton buffer.          *)
    iindex    : INTEGER ;         (* Current daton index.                      *)
                       (* iindex = index of next daton to be received.     *)
                       (*        = index+1 of daton in tailcell.           *)
    active    : INTEGER ;         (* Number of outports not finalindex         *)
    profiting : INTEGER ;         (* Number of outports with novalues=0        *)
    customers : INTEGER ;         (* Number of outports with:                  *)
                                  (* outpool[i].waiting = TRUE                 *)
    p0        : ACTOR   ;         (* Operand actor.                            *)
    semaphore : ACTOR   ;
    noutports : INTEGER ;         (* Number of outports.                       *)
    outpool   : ARRAY [1..noutports] OF OUTPORTSTRUCT ;
   END ;
```

The current *I*nport index `iindex` refers to the daton ***presently due*** to reach the inport, the current *O*utport index `oindex` refers to the daton presently due to come out of the respective outport. (A real programming language would hardly permit a ***dynamic*** array as an element of a data record, such as `outpool` above. However, every implementor knows alternative ways for achieving the same effect.) In procedure headings, we will repeatedly encounter formal parameters of the kind:

`VAR outport : OUTPORTSTRUCT`

When looking up the corresponding actual parameters it will always turn out that `outport` is merely an alias for `outpool[i]`, which in turn is an array element within `INPORTSTRUCT`; *i* is outport dependent.

The `INPORTSTRUCT` and all the various `OUTPORTSTRUCT` of the entire `COPY` node actor get initialised when the inport actor calls the procedure `InitialiseCopy`:

```
PROCEDURE InitialiseCopy (netcreator : ACTOR ;
                          VAR inport : INPORTSTRUCT) ;
   VAR i : INTEGER ;
   BEGIN WITH inport DO
   BEGIN
     profiting := noutports ;      customers := 0 ;
     active    := noutports ;      iindex    := 0 ;
     tailcell  := NIL        ;
     semaphore := CREATE (Act_Guardian_) ;

     FOR  i:=1 TO noutports
     DO WITH outpool[i] DO
        BEGIN
          oactor  := CREATE (Act_CopyOutport_ ,
                             inport, outpool[i]) ;
          SEND (DATON, oactor) TO (netcreator)  ;
          buffer  := NIL   ;      oindex   := 0 ;
          waiting := FALSE ;      novalues := 0 ;
        END ;

     ( , , p0) := RECEIVE FROM (netcreator) ;
   END END ;              (* InitialiseCopy *)
```

4.6.4 Request Propagation, and Voting

We mentioned in section 4.2 the two diametrically opposed strategies which govern the propagation of requests. COPY issues a COMPUTE request whenever ***any*** of its outports needs the daton value without the daton having been buffered yet. After the COMPUTE, a counteracting NULLIFY may be sent if the daton evaluation proves superfluous. COPY sends an ADVANCE as soon as it has accepted the daton value for the daton buffer.

On the other hand, an outport can get many bare ADVANCE requests in a row. Such requests may eventually put the outport into the ***off-chain state***. We like to propagate ADVANCE requests, in general, at the earliest possible moment, since they are capable of releasing buffer space in COPY nodes "further upstream" in the Lucid graph. However, any ADVANCE can be propagated only if there will be definitely no subsequent demand for the current daton. COPY can therefore propagate ADVANCE only when (the daton buffer is empty and) each outport has surrendered its claim for the current daton. Each time COPY obtains a new daton (and its cell has been appended to the tail end of the chain) it checks whether the chain has now caught up with any of the off–chain outports. If appropriate, the outport is then ***hooked in*** at the chain tail.

Every ADVANCE, finalindex puts the outport into **finalindex–state**, and the finalindex–state implies off–chain state. Once an outport enters the finalindex–state it withdraws ***all*** further claims to datons. All the rules about ADVANCE have to be extended accordingly to this special ADVANCE. The effect of COPY receiving an ADVANCE, finalindex is usually tantamount to receiving infinitely many bare ADVANCE. Occasionally, it may lead to the propagation of ***many*** ADVANCE requests; this would be due to the WHILE loop in IncrementNovalues.

The ADVANCE propagation is implemented in COPY by what is essentially a **vote counting** where all decisions have to be unanimous. Each outport records in a cell (named novalues) by how many datons it has advanced beyond the current inport daton. The inport records in a cell profiting how many of its outports might benefit from knowing the value of the current daton, i.e. how many of its outports have novalues = 0. So, profiting is decremented whenever an outport increases its novalues from 0 to 1, and vice versa. Once all novalues are greater than zero (i.e. once profiting = 0) an ADVANCE can be propagated to the operand actor. After every increment of index, like now, all positive novalues can be decremented. Most of what is described in this paragraph is carried out by the procedure IncrementNovalues (↑ 4.6.7). The procedure DecrementNovalues performs obviously the inverse task.

4.6.5 Despair and the "Trojan Horse"

The COPY node actor propagates, by design, only the ***least expensive*** request for getting the job done. However, situations can arise where wasteful computations are hard to avoid in pipeline DF. Let us consider a COPY node with 2 active outports named *X* and *Y*, and we are at the beginning of program execution. Outport *Y*

receives a bare $\boxed{\text{ADVANCE}}$, but outport *X* receives no request yet. The $\boxed{\text{novalues}}$ of *Y* is now 1, the $\boxed{\text{novalues}}$ of *X* remains 0. Next, *Y* gets a $\boxed{\text{COMPUTE}}$ request. We cannot simply skip the daton at index 0 and evaluate daton 1, since we do not know whether *X* will eventually ask for the value of daton 0, and pipeline DF allows only the daton evaluation in the order of increasing index. Out of "despair", we have to evaluate daton 0 and queue it in outport *X*. The evaluation of daton 0 will have been in vain if *X* then chooses to start with a *bare* $\boxed{\text{ADVANCE}}$ request. Such a situation would be handled much more efficiently in a *tagged DF* implementation.

We can give this example a different twist. We can assume that the evaluation of daton 0 takes a day (or it may take ***forever***), and that *X* gets a bare $\boxed{\text{ADVANCE}}$ after the ***first*** second into this evaluation. The operand actor must immediately be given a $\boxed{\text{NULLIFY}}$, since the evaluation is now clearly unwanted. This means that even if only $\boxed{\text{ADVANCE}}$ and $\boxed{\text{COMPUTE}}$ requests are ever issued to the $\boxed{\text{COPY}}$ node actor it must be permitted to generate $\boxed{\text{NULLIFY}}$ requests of its own accord. In other words, the implementation (pipeline DF) would be ***incomplete*** without $\boxed{\text{NULLIFY}}$.

This constellation of requests is about the evaluation of a daton which no outport really wants; the daton is a "**Trojan Horse**". We will come back to it when studying the inport act.

4.6.6 An Invariant

Using ***ql*** to denote the ***q***ueue ***l***ength (the number of buffer cells on the ***tail*** side of the buffer pointer), the following holds for every outport:

```
as long as    oindex <> finalindex  then:
   0 = ql - novalues + oindex - iindex     (invariant)
   0 = ql * novalues

ql, novalues, oindex and iindex are all
                                  non-negative integers.
```

4.6.7 Procedures for COPY Outport Act

The concepts underlying the procedures DecrementNovalues and IncrementNovalues have been explained in the subsection ***request propagation*** above.

```
PROCEDURE DecrementNovalues (VAR  inport  :  INPORTSTRUCT ;
                             VAR outport  : OUTPORTSTRUCT ) ;
   BEGIN WITH inport, outport DO BEGIN
          novalues  := novalues  - 1 ;
     IF   novalues   = 0
     THEN profiting := profiting + 1 ;
   END END ;
```

```
PROCEDURE IncrementNovalues (VAR  inport  :  INPORTSTRUCT ;
                             VAR outport  : OUTPORTSTRUCT ) ;
   VAR   i : INTEGER ;
   BEGIN WITH inport, outport DO
   BEGIN
     IF   novalues   = 0
     THEN profiting := profiting - 1 ;
     novalues       := novalues  + 1 ;

     WHILE profiting = 0
     DO BEGIN
          EXCEPTION ADVANCE TO (iactor) ;
          FOR  i:=1  TO noutports
          DO   IF   outpool[i].oindex <> finalindex
               THEN DecrementNovalues (inport, outpool[i]) ;
        END ;
   END END ;
```

The procedure AdvanceBufferPointer, below, advances (by one daton) the buffer pointer of an outport. The ***reference count*** allows us to decide when a buffer cell can be freed. The cell can be freed only if it is certain that the daton value will ***never*** be needed again (old reference count = 1).

```
PROCEDURE AdvanceBufferPointer (VAR outport : OUTPORTSTRUCT) ;
   VAR
     oldcell : CELLP ;     h : INTEGER ;
   BEGIN
     oldcell         := outport.buffer ;
     h               := oldcell^.count ;
     outport.buffer  := oldcell^.link   ;   (* This can be NIL. *)
     IF   h = 1
     THEN FreeCell (oldcell)
     ELSE oldcell^.count := h - 1 ;
   END ;
```

The procedure $\boxed{\text{AdvanceOutport}}$ takes care of the entire $\boxed{\text{ADVANCE}}$ handling of the $\boxed{\text{COPY}}$ outport. It resolves every ordinary $\boxed{\text{ADVANCE}}$ request by calling either $\boxed{\text{IncrementNovalues}}$ or $\boxed{\text{AdvanceBufferPointer}}$. However, the full $\boxed{\text{ADVANCE}}$ handling requires more than that. An $\boxed{\text{ADVANCE, finalindex}}$ request puts one outport into the finalindex–state ($\boxed{\text{oindex}}$ = $\boxed{\text{finalindex}}$). We must in this case check first if there is an outport left which is not in finalindex–state. This check is done by a vote counting. The inport records in a cell, named $\boxed{\text{active}}$, how many of its outports are still ready to transport datons, i.e. not in finalindex–state. Once all outports are $\boxed{\text{finalindex}}$ (i.e. once $\boxed{\text{active}}$ = 0), an $\boxed{\text{ADVANCE, finalindex}}$ can be propagated to the operand actor. However, if there are *active* outports left, $\boxed{\text{IncrementNovalues}}$ must be carried out even upon the arrival of an $\boxed{\text{ADVANCE, finalindex}}$ request at the outport.

```
PROCEDURE AdvanceOutport (VAR  inport  :  INPORTSTRUCT ;
                          VAR outport  : OUTPORTSTRUCT ) ;
   VAR   request : MSGTYPE ;   index : INTEGER ;
   BEGIN WITH inport, outport DO
   BEGIN
     MakeExclusive (semaphore) ;
     (request, index) := Reveal ;

     IF   index = finalindex
     THEN BEGIN
            oindex := finalindex ;

            WHILE buffer <> NIL
            DO    AdvanceBufferPointer (outport) ;

            active := active - 1 ;
            IF   active = 0
                 (* There is no need to bother the inport actor.  *)
                 (* iindex := finalindex ;      not essential     *)
            THEN EXCEPTION (ADVANCE, finalindex) TO (p0)
            ELSE IncrementNovalues (inport, outport) ;
          END

     ELSE BEGIN
            oindex := oindex + 1 ;
            (* IF  oindex <> index  THEN ReportError ;            *)

            IF   buffer = NIL
            THEN IncrementNovalues (inport, outport)
            ELSE AdvanceBufferPointer (outport) ;
          END ;

     MakePublic (semaphore) ;
   END END ;            (* End of AdvanceOutport .                *)
```

4.6.8 COPY Outport Act

Here is the act of a single COPY outport:

```
ACT Act_CopyOutport_ (VAR  inport  :  INPORTSTRUCT ;
                      VAR outport : OUTPORTSTRUCT ) ;
   LABEL 1, 2, 3, 4, 5, 6 ;
   VAR sender    : ACTOR    ;     (* Temporary variable.                  *)
       dvalue    : ANYTYPE ;      (* Reply daton value.                   *)
       superior  : ACTOR    ;     (* Request sender.                      *)
       request   : MSGTYPE ;      (* Incoming request.                    *)
       index     : INTEGER ;      (* Index in the incoming request.       *)
   BEGIN WITH inport, outport DO
   BEGIN

   REPEAT
     WHILE TRUE DO
     BEGIN                                                                  :6
        (superior, request, index) := RECEIVE () ;
        (* IF   index <> oindex THEN ReportError ;                        *)
                                                                            :6
        MakeExclusive (semaphore) ;                                         :5
        IF   buffer <> NIL                                                  :5
        THEN MakePublic (semaphore)      (* i.e. go right ahead.           *)

        ELSE BEGIN
               waiting   := TRUE ;                                          :4
               IF   customers = 0  THEN                                     :3
                    SEND COMPUTE TO   (iactor) ,    (* Activate.           *)
               customers := customers + 1 ,                                 :2
               MakePublic (semaphore) ;                                     :1
               sender := RECEIVE FROM (iactor) .    (* Wait.               *)
             END ;
                                                                            :6
        dvalue :=  buffer^.value ;                                          :6
        SEND (DATON, dvalue) TO (superior) ;
     END ;              (* End of the inner eternal loop.                  *)

   (* Exception part.                                                      *)
1:   MakeExclusive (semaphore) ;
       IF   waiting
       THEN BEGIN
2:              customers      := customers - 1 ;
                IF   customers = 0
                THEN
3:                   EXCEPTION NULLIFY TO (iactor) ;
4:              waiting := FALSE ;
             END ;
5:   MakePublic (semaphore) ;

6:   IF   Reveal = ADVANCE
     THEN AdvanceOutport (inport, outport) ;
     RESET ;
  UNTIL FALSE ;        (* End of the exception handling loop.             *)
  END END ;            (* End of Act_CopyOutport_ .                       *)
```

```
ACT Act_CopyOutport_ (VAR  inport  :  INPORTSTRUCT ;
                      VAR outport : OUTPORTSTRUCT ) ;
   LABEL 1, 2, 3, 4, 5, 6 ;
   VAR sender    : ACTOR   ;     (* Temporary variable.                 *)
       dvalue    : ANYTYPE ;     (* Reply daton value.                  *)
       superior  : ACTOR   ;     (* Request sender.                     *)
       request   : MSGTYPE ;     (* Incoming request.                   *)
       index     : INTEGER ;     (* Index in the incoming request.      *)
   BEGIN WITH inport, outport DO
   BEGIN

   REPEAT
     WHILE TRUE DO
     BEGIN                                                                 :6
        (superior, request, index) := RECEIVE () ;
        (* IF   index <> oindex THEN ReportError ;                     *)
                                                                           :6
        MakeExclusive (semaphore) ;                                        :5
        IF   buffer <> NIL                                                 :5
        THEN MakePublic (semaphore)      (* i.e. go right ahead.       *)

        ELSE BEGIN
               waiting    := TRUE ;                                        :4
               IF   customers = 0  THEN                                    :3
                    SEND COMPUTE TO   (iactor) ,     (* Activate.      *)
               customers := customers + 1 ,                                :2
               MakePublic (semaphore) ;                                    :1
               sender := RECEIVE FROM (iactor) ;     (* Wait.          *)
             END ;
                                                                           :6
        dvalue :=  buffer^.value ;                                         :6
        SEND (DATON, dvalue) TO (superior) ;
      END ;               (* End of the inner eternal loop.            *)

   (* Exception part:                                                  *)
1:    MakeExclusive (semaphore) ;
         IF   waiting
         THEN BEGIN
2:                customers     := customers - 1 ;
                  IF   customers = 0
                  THEN
3:                     EXCEPTION NULLIFY TO (iactor) ;
4:                waiting := FALSE ;
                END ;
5:    MakePublic (semaphore) ;

6:    IF   Reveal = ADVANCE
      THEN AdvanceOutport (inport, outport) ;
      RESET ;
   UNTIL FALSE ;        (* End of the exception handling loop.         *)
   END END ;            (* End of Act_CopyOutport_ .                   *)
```

The $\boxed{\text{COPY}}$ outport actor enters an eternal loop right away. Each loop pass starts with the acceptance of a $\boxed{\text{COMPUTE}}$ request. The validity of the daton index can be checked here, an error would be a ***system*** error. $\boxed{\text{COMPUTE}}$ is trivial to handle if the wanted daton is ready waiting in the buffer; the daton is simply taken from the buffer and sent to the superior. If, however, the buffer is found empty, the daton evaluation must be instigated, its outcome must be waited for, and only then can the reply be given to the superior.

Further ***vote counting*** is used to control the inport. The cell $\boxed{\text{customers}}$ states how many outports are hung up waiting for the arrival of the next daton. Outports increment this cell when appropriate, and ***send*** an activating signal to the inport whenever incrementing $\boxed{\text{customers}}$ from 0 to 1. Further increments require no signalling to the inport since it ***is*** already busy with the evaluation. However, outports are free to withdraw their demands at any time, and they do this by ***decrementing*** $\boxed{\text{customers}}$. A $\boxed{\text{NULLIFY}}$ is sent to the inport actor right after $\boxed{\text{customers}}$ has been decremented to 0.

Before an outport gets hung up waiting for the daton, it sets furthermore its $\boxed{\text{waiting}}$ flag. The inport is thus able to identify every demanding outport. When the daton arrives (via $\boxed{\text{GetDaton}}$), the $\boxed{\text{COPY}}$ inport sends a releasing signal to each ***waiting*** outport. (The execution of faulty Lucid programs can easily seize up in a Deadlock, † 2.6 and 6.1. In such a case, the outport actor will hang up waiting ***forever*** for the daton. This error can be detected ***automatically*** by the message passing mechanism.)

4.6.9 $\boxed{\text{COPY}}$ Outport Exception Handling

The action in the event of a $\boxed{\text{NULLIFY}}$ exception depends on the stage the daton evaluation has reached. If the $\boxed{\text{NULLIFY}}$ occurs ***after*** the arrival of the daton at the outport actor, the exception has ***no*** genuine effect. However, the shorter the exception occurs *before* that moment, the more preparations for the daton delivery have been undertaken, each of them needing to be reversed. The code for handling NULLIFY exceptions is therefore almost a mirror–image of the preceding code. Upon exception, execution jumps from one instruction to its counterpart and reverses each preparation in turn. The NULLIFY request sent to the inport actor counteracts its preceding COMPUTE.

ADVANCE can be described as an *extension* of this NULLIFY. If an ADVANCE exception did occur during daton evaluation it would have to start with the action for a NULLIFY exception. (In real life, ADVANCE exceptions do not occur while the outport is waiting for a daton.) ADVANCE exceptions are handled by the procedure AdvanceOutport (↑ 4.6.7).

We turn our attention now from the COPY outport to the COPY *inport*.

4.6.10 Procedure for COPY Inport Act

Before we deal with Act_Copy_ (the COPY *inport* act) we study its special procedure UpdateOutports. Whenever the inport receives a daton value (via GetDaton), it puts it into the daton buffer. This puts the outports into a totally new situation, even the invariant is corrupted, and corrective action is necessary for most outports. The procedure UpdateOutports contains all this action.

Let us assume, a daton had just arrived. Outports in finalindex–state require no action, nor do outports with datons *queued*. Outports with novalues > 0 have to decrement it by one (DecrementNovalues takes care of this). All remaining outports must be linked to the tail of the daton chain. Every *waiting* outport among them needs an update of customers and waiting, and a reactivating signal must also be sent to it.

```
PROCEDURE UpdateOutports (VAR inport : INPORTSTRUCT) ;
   VAR     i : INTEGER ;
   BEGIN   WITH inport DO
   BEGIN
   FOR     i:=1 TO noutports  DO
    WITH   outpool[i] DO
     IF    (oindex <> finalindex)  AND  (buffer = NIL)  THEN
      BEGIN
      IF   0 < novalues
      THEN DecrementNovalues (inport, outpool[i])
      ELSE BEGIN
             buffer        := tailcell ;
             buffer^.count := buffer^.count + 1 ;
             IF   waiting
             THEN BEGIN         (* reactivate outport:          *)
                    customers := customers - 1 ;
                    waiting   := FALSE          ;
                    SEND DATON TO (oactor)      ;  (* Release.  *)
      END  END     END
   END  END ;                   (* End of UpdateOutports .      *)
```

4.6.11 COPY Inport Act

In our Sieve example (section 4.3.3.1, Act_Sieve), a COPY node actor with 4 outports is set up by the LUX instructions:

```
node [2] := CREATE (Act_Copy_ , 4) ;
( , , node [1]) := RECEIVE FROM (node[2]) ;
( , , node [5]) := RECEIVE FROM (node[2]) ;
( , , node [8]) := RECEIVE FROM (node[2]) ;
( , , node[10]) := RECEIVE FROM (node[2]) ;
 ...
SEND (DATON, operand_actor) TO (node [2]) ;
```

We created only the *inport* actor of COPY (i.e. node[2]), and it created the outport actors of its own accord, though telling us their actor names. We sent the initialisation to the inport; the inport itself looked after its linkage with the outports, and *their* initialisation.

So, here is the LUX code for the universal multi-outport COPY node (using pipeline demand driven DF) or just the COPY *inport* act, depending on your point of view:

```
ACT Act_Copy_ (n : INTEGER) ;
   LABEL 1 ;                       (* n-outport COPY node (inport).       *)
   VAR
     newcell : CELLP    ;          (* Temporary variable.                 *)
     sender  : ACTOR    ;          (* Temporary variable.                 *)
     dvalue  : ANYTYPE ;           (* Daton value received from operand.  *)
     inport  : INPORTSTRUCT ;      (* Characterisation of this inport.    *)
   BEGIN WITH inport DO
   BEGIN
     iactor := Myself ; noutports := n ; InitialiseCopy (Creator, inport) ;

   REPEAT
     WHILE TRUE DO
     BEGIN
        IF  customers = 0  THEN                                            :1
            sender := RECEIVE () ;      (* Wait for Activation.      *)
                                                                           :1
        dvalue := GetDaton (iindex, p0) ;

        newcell         := GetCell ;     (* GetCell can take long.    *)
        newcell^.link   := NIL     ;
        newcell^.value  := dvalue  ;

        MakeExclusive (semaphore) ;      (* Applied as late as possible. *)

        IF   Reveal = ADVANCE          (* Test for "Trojan Horse".   *)
        THEN FreeCell (newcell)
        ELSE BEGIN
                iindex := iindex + 1 ;
                EXCEPTION (ADVANCE, iindex) TO (p0) ;
                  IF      tailcell = NIL
                  THEN     newcell^.count := 0
                  ELSE BEGIN
                           newcell^.count := tailcell^.count ;
                          tailcell^.link  :=  newcell        ;
                       END ;
                  tailcell := newcell    ;
                UpdateOutports (inport) ;
             END ;

        MakePublic (semaphore) ;
      END ;          (* End of the eternal loop.                  *)

    (* Exception handling: *)
1:  IF   Reveal = ADVANCE
    THEN BEGIN
           iindex := iindex+1 ; EXCEPTION (ADVANCE, iindex) TO (p0) ;
         END ;
    RESET ;
    UNTIL FALSE ;     (* End of the exception handling loop.          *)
  END END .           (* End of Act_Copy_ .                           *)
```

The COPY inport actor is not a ***node*** actor, i.e. it does not accept ***requests***. It exchanges merely ***signalling messages*** with each outport (however, the communication with its ***operand*** adheres entirely to the request protocol). Every signal from the inport to an outport is of message type DATON, just as an indication that this is neither a request nor an exception.

The inport actor owns (declares) and initialises all the descriptors relating to this COPY. The inport descriptor contains all the outport descriptors. InitialiseCopy contains almost all the initialising action. It passes the names of all the outport actors to the creator of the computing net, and it acquires finally the name of the operand actor.

After the initialisation, the inport actor enters an eternal loop. The loop starts with a RECEIVE, which serves a ***similar*** purpose as the request RECEIVE in node actors. As long as ***no*** outport is waiting for a daton, the inport actor becomes dormant until an outport spurs it into action by sending a signal. This signal means invariably "evaluate the current daton". The daton value is acquired from the operand actor (via GetDaton), an ADVANCE is issued to the operand actor ***right away***, and the daton is appended to the daton chain. The full benefit of the new daton is then given to the outports through calling UpdateOutports.

4.6.12 Exceptions Sent by COPY Inport

In their internal communication, the COPY inport and its outports do not view each other as ***node*** actors, and do therefore not follow the universal protocol. However, we employ most of the exception mechanism even then; the index field is not used. The exception part of the inport act is simple.

Let us first concentrate on NULLIFY exceptions. Above, we have described the customers voting mechanism. The inport gets the NULLIFY exception whenever customers drops from 1 to 0. The GetDaton propagates NULLIFY exceptions to the operand actor if necessary. If the exception arrives ***after*** the daton has arrived in $\boxed{\text{GetDaton}}$, the inport will at first not react to the exception but will buffer the daton properly. Wasteful re–computation of the daton is avoided by this ***eager buffering***.

It has been mentioned that an acknowledging $\boxed{\text{ADVANCE}}$ is automatically issued by the inport actor ***right after*** the acceptance of each daton value. Whenever the inport gets an $\boxed{\text{ADVANCE}}$ exception, this can only be due to a bare $\boxed{\text{ADVANCE}}$, or to $\boxed{\text{ADVANCE, finalindex}}$ at one of the outports. The propagation of a bare $\boxed{\text{ADVANCE}}$ is the aim in either case. However, a daton evaluation may be under way (in the operand actor) while the exception occurs, i.e. we find ourselves in the "***Trojan Horse***" situation. The evaluation must in this case be nullified. If the daton has already been accepted, there is no point in buffering it. Finally, an $\boxed{\text{ADVANCE}}$ is propagated.

Usually, it is the ***inport*** actor of $\boxed{\text{COPY}}$ which issues the requests to the operand actor. However, $\boxed{\text{ADVANCE, finalindex}}$ is different in being issued directly by a $\boxed{\text{COPY}}$ outport actor. This cannot lead to a ***collision*** with requests from the inport actor, since (as a precondition) all outports will be in finalindex–state anyway, and the inport will therefore be dormant. The semaphore keeps the outports from issuing colliding requests. The inport circumvention is therefore permissible in this case.

4.6.13 Concurrency in $\boxed{\text{COPY}}$

One might ask what gives us the right to call this $\boxed{\text{COPY}}$ act ***concurrent***. Restrictions of concurrency are hard to accept if no valid reasons can be given.

Concurrency means ***simultaneous action in various places***. During the execution of a Lucid program we associate computing action with every node in the Lucid graph. In demand driven evaluation, this action is restricted to those nodes whose output is ***essential for the result presently due***. We chose a ***version*** of demand driven evaluation where, at any time, solely the ***current*** result daton is in evaluation (or ***contributing*** datons). The alternative, "**bulk demand**" (e.g. "give me the next 100 operand actor if necessary. If the exception arrives ***after*** the daton has arrived in $\boxed{\text{GetDaton}}$, the inport will at first not react to the exception but will buffer the daton properly. Wasteful re–computation of the daton is avoided by this ***eager buffering***.

It has been mentioned that an acknowledging $\boxed{\text{ADVANCE}}$ is automatically issued by the inport actor ***right after*** the acceptance of each daton value. Whenever the inport gets an $\boxed{\text{ADVANCE}}$ exception, this can only be due to a bare $\boxed{\text{ADVANCE}}$, or to $\boxed{\text{ADVANCE, finalindex}}$ at one of the outports. The propagation of a bare $\boxed{\text{ADVANCE}}$ is the aim in either case. However, a daton evaluation may be under way (in the operand actor) while the exception occurs, i.e. we find ourselves in the "***Trojan Horse***" situation. The evaluation must in this case be nullified. If the daton has already been accepted, there is no point in buffering it. Finally, an $\boxed{\text{ADVANCE}}$ is propagated.

Usually, it is the ***inport*** actor of $\boxed{\text{COPY}}$ which issues the requests to the operand actor. However, $\boxed{\text{ADVANCE, finalindex}}$ is different in being issued directly by a $\boxed{\text{COPY}}$ outport actor. This cannot lead to a ***collision*** with requests from the inport actor, since (as a precondition) all outports will be in finalindex–state anyway, and the inport will therefore be dormant. The semaphore keeps the outports from issuing colliding requests. The inport circumvention is therefore permissible in this case.

4.6.13 Concurrency in $\boxed{\text{COPY}}$

One might ask what gives us the right to call this $\boxed{\text{COPY}}$ act ***concurrent***. Restrictions of concurrency are hard to accept if no valid reasons can be given.

Concurrency means ***simultaneous action in various places***. During the execution of a Lucid program we associate computing action with every node in the Lucid graph. In demand driven evaluation, this action is restricted to those nodes whose output is ***essential for the result presently due***. We chose a ***version*** of demand driven evaluation where, at any time, solely the ***current*** result daton is in evaluation (or ***contributing*** datons). The alternative, "**bulk demand**" (e.g. "give me the next 100

datons"), is at present too hard to solve in general. Once committed to the daton–by–daton approach, our sequential request protocol brings no new restriction.

Whenever a $\boxed{\text{COMPUTE}}$ request is sent to a $\boxed{\text{COPY}}$ outport, the daton delivery may be held up until the daton value has arrived at the $\boxed{\text{COPY}}$ inport. This restriction comes from causality, it cannot be defeated. All other requests are accepted and handled without major delay. Occasionally, a $\boxed{\text{COPY}}$ outport may be shortly hung waiting for the completion of action by other actors; $\boxed{\text{GetCell}}$ is probably the worst source of delay. The semaphore, in particular, forces potentially conflicting actions into sequential order. Each $\boxed{\text{COPY}}$ outport can handle any request which satisfies the protocol (↑5.5). Its freedom of choice is never dependent on states of other outports.

4.6.14 Summary of $\boxed{\text{COPY}}$ Act

What we have just described is the universal $\boxed{\text{COPY}}$ act. It is so complicated because it caters for every possible situation (within demand driven pipeline DF). Whenever more is known about the way in which the $\boxed{\text{COPY}}$ node actor is to be used, this extra information can be put to good use. In such cases, it may be possible to use a much *simpler* $\boxed{\text{COPY}}$ act. Is anything known about the order in which the requests arrive at the outports? Is anything known about the maximum queue length? Do we really ever request concurrently? Chapter VI, "*Efficiency*", will present specialised versions of $\boxed{\text{COPY}}$. Before that, chapter V will show a method for checking the correctness of the $\boxed{\text{COPY}}$ act. In doing this, chapter V will also give a second description of how $\boxed{\text{COPY}}$ works: this may help to clear up remaining points of uncertainty.

4.7 Priority Scheduling

4.7.0 Introduction

So far, we have learnt how to translate a Lucid program into message passing actors. Every instance of an operator (including any UDF) maps into an actor. The resulting number of actors is extremely high, judging by the standards of current multi–process operating systems. Highly concurrent computation in many actors, however, is just the thing which the newest generation of computers (vast numbers of physical processors) is best suited for. This thesis will not even try to answer the specific questions coming with ***multi***–processor implementations of Lucid, such as:

- What is the best strategy for allocating and scheduling the multitude of actors on a ***smaller*** number of processors?
- For recursive (dynamically expanding) UDFs, how and where are the new actors allocated?

The answers to these questions depend much on properties of the given hardware: the store structure (shared or dedicated), availability of virtual store, availability of runtime load, etc.

On the other hand, readers who wish to do a serious implementation of Lucid on a conventional computer (von Neumann ***mono***processor) will be relieved to hear that chapter VI will show how the number of actors can be reduced towards more acceptable bounds. There is no reason why it should be impossible to compile into a ***single*** actor any Lucid program without concurrent operators (parallel OR, etc.) and without recursive UDFs, i.e. compile it into a conventional ***sequential*** program. However, the ***general*** algorithm for that reduction is yet to be invented. At least up to that day, we need a rule for ***scheduling*** the actors. (At any moment, only ***one*** actor can be in actual execution. The **scheduling** rule states ***which*** actor to execute, and ***for how long***.)

We present in this section an actor scheduling rule based on ***priorities***. The rule may be far from optimal, but it will be sufficient to achieve a reasonably well balanced program execution. (The rule is aimed at granting, to an evaluation, resources in proportion to the relevance of its result.) This topic will not be treated exhaustively in this thesis; merely a few guidelines will be presented.

4.7.1 Analogies

We can draw a parallel between the execution of a program Luprog and the running of a (somewhat strange) firm for technical developments, "Luprox Ltd". Some company workers develop one entire product after another, while others carry out only partial production steps and have to cooperate with others. Occasionally, the manager chooses to let separate (groups of) workers develop competing products. Sometimes he uses everything that emerges from this concurrency. In some cases a production order is cancelled or a product is thrown away because it has become superfluous. Each department is run as an autonomous unit, but the management policy is identical on each level. The investment policy is somewhat simple minded: whenever concurrent developments are instigated, each development gets an ***equal*** share of the departmental resources.

If any department requires two equal ranking concurrent sub–developments, the department dedicates half of its capacity to each of them. If either of the resulting ***sub***–departments needs to break its work into 3 sub–sub–developments, the capacity of the sub–department is split into three equal parts, and each of the sub–sub–departments gets 1/6 of the original capacity. – On the other hand, if a department works for a number of other departments (as in the case of the hardware store, or catering) it has the ***sum*** of its user's allocations as funds.

It has been decided that a more refined management policy would require an inappropriately expensive case analysis. Indeed, there is only one manager in the company, and each department calls him in for administration. Worse even, the company is a one–man business where one man is playing all the roles in turn, be it manager or be it any worker. He is obviously not ready to spend much time on administration.

The management policy (scheduling, resource allocation) must be applied with flexibility, since the assumptions on which it is based are so imprecise. It is mainly designed to make sure that all the work will eventually be done, and that not too much of the resources are wasted on work of low importance.

This management policy may be bearable for the one man business, but it is really too vague for a company with *many* workers. For example, for which job shall each worker be trained? Moreover, the manager insists on maintaining the correct sequence of product delivery purely by the sequence of the work pieces on the conveyer belts (pipeline DF). Workers must therefore never *share* jobs.

We could even link this analogy with our earlier analogy in chapter III. The example above might describe the management policy of a restaurant "*Chez Lucien*" which is run by one man alone: waiter, cook and manager in one person. The scheduling rule tells him, for example, in which order to prepare the meals for his customers, even in which order to bother about courses and parts of each course.

Now replace *CPU* for our busy Jack of all trades, *runtime system* or *scheduler* for manager, and *actor* for department or sub*–department. The company carries out computations to order; the products (developments) of the company are the datons of the program's result. The total production capacity of the company is determined by the power of the given CPU.

4.7.2 Our Scheduling Rule

Our scheduling rule deals only with one computing resource, namely ***CPU time***. We aim to be reasonably fair in sharing out the available computing capacity, and we use the ***subdividing rule*** just described. If we define the total available computing capacity as "1", we can use ***fractions*** to express how big a "***capacity slice***" each actor gets, i.e. the **priority** of each actor. An actor which has 1/2 of the capacity allocated obviously gets through its work more rapidly than an actor with 1/9.

Our scheduling rule distinguishes different kinds of priority. Every actor has its specific priorities (stored within the actor head, ↑ 3.2.1), and the really decisive one among them is the ***actual priority***. The actual priority is calculated, among other things, from the ***intrinsic priority***. The intrinsic priority is, generally speaking, an actor specific constant:

actor	intrinsic priority	
root actor	–	"ultra"
WRITE node actor	1.0	top (is explicitly set by creator)
all other node actors	0.0	zero (default, meaning: DON'T EXECUTE)

The actor's **actual priority** fluctuates with its message passing state, in detail:

```
actual_priority  :=
     IF        xrequest <> READY
     THEN      ultra_priority
     ELSE IF ( some actors are hung trying to send
               to this one, or to receive from it )
        THEN ( sum of their shared out actual priorities )
        ELSE   intrinsic_priority ;
```

The scheduler has (at least) ***two*** queues for actors: the **ultra queue** of actors with **ultra priority** and the **normal queue** for all remaining actors with actual priority not zero. Actors in the normal queue are executed only if the ultra queue is empty. "***First come first serve***" and "***round robin***" apply inside each queue. Actors in the ultra queue are executed to exhaustion, i.e. control is taken from them only as late as possible. If not in ultra priority, an actor is treated as **normal**. The normal actors share the computing resources (mainly: the ***time*** spent in execution) in proportion to their ***actual priorities***. An actor is ***suspended*** from execution while hung waiting for message passing, of course.

By "***sharing out***" we mean: if an actor is hung trying to $\boxed{\text{SEND}}$ to, or to $\boxed{\text{RECEIVE}}$ from, a set of actors, it shares its own actual priority out in equal parts among the actors it wants to communicate with. However, $\boxed{\text{ultra}}$ divided by ***any*** number is still $\boxed{\text{ultra}}$, and $\boxed{\text{ultra}}$ plus anything is $\boxed{\text{ultra}}$.

When determining the actual priority of node actor *Z* we may have to form the ***sum*** of some shared out priorities. In this sum, we must exclude any contribution which is due to ***Z itself*** (indirectly). – Such a "priority sum" needs to be formed only if *Z* is a $\boxed{\text{COPY}}$ node actor; without this exclusion rule, cycles could "hype up" their own priority. The scheduler should even issue a $\boxed{\text{NULLIFY}}$ request to the $\boxed{\text{COPY}}$ inport actor whenever the actual priority of the $\boxed{\text{COPY}}$ falls to zero. The scheduler introduces thus a measure of ***global control***, which would be impossible to achieve by the request protocol alone (we will touch a similar point at the end of 6.3).

Equivalent to the if–then–else rule above, an actor's actual priority can be calculated as the ***maximum*** of:

(1) its intrinsic priority,

(2) $\boxed{\text{ultra}}$ priority while its $\boxed{\text{xrequest}}$ <> $\boxed{\text{READY}}$,

(3) the sum of the ***shared out*** actual priorities of all actors which are currently hung up waiting for communication with the actor in question ($\boxed{\text{SEND}}$ to it or $\boxed{\text{RECEIVE}}$ from it).

The following can be deduced from the scheduling rule:

– The actual priority of a $\boxed{\text{COPY}}$ inport actor is the sum of the actual priorities of its waiting outports (though not forgetting the exclusion rule).

– An inferior will not be executed unless its superior gets hung up.

4.7.3 Discussion of Scheduling Rule

The scheduling rule contains nothing to prevent an actor from ***livelocking*** (↑ 2.6). A livelocking actor with ultra priority would be total disaster since it would never surrender its execution right. Our design of the individual acts, however, makes sure that only ***finite*** computations are ever undertaken in ultra priority. The ultra priority has actually been invented exclusively for urgent administration and for nullification of unwanted evaluations.

Also by design, there are very few instances where an actor has more than one actor trying to communicate with it (semaphores, and arrival of concurrently evaluated datons are obvious instances). The *FCFS* strategy and, if necessary, random sequence are sufficient to ensure correct behaviour. (Easy evaluations will usually succeed before elaborate ones, this is important for concurrent OR etc. Our scheduling rule executes concurrent operations in the "***breadth first***" strategy.)

The design of the *UDF* actors is particularly tuned for this priority mechanism. UDF actors have an intrinsic priority of zero, and this makes sure that execution of the UDF ***pauses*** before the subnet creation (= expansion). The expansion is carried out once the UDF gets its first request; this is ***lazy*** expansion. In effect, the subnet actors are created as ***late*** as possible. Initial ADVANCE or ADVANCE, finalindex requests needed special treatment; this ensures that only short administration is ever undertaken in ultra priority (but not ***proper*** computation, or even subnet creation).

Our scheduling rule is open to much criticism. For example, we assumed that the subcomputations of one computation are ***equivalent***, which can be easily disproved by counter examples. However, our rule is reasonable and cheap. Indeed, a better rule can not be provided if ***nothing*** is known about internals of the actors (e.g. if it unknown how important a particular computation is). — We have already stated that the priority concept provides only an incomplete answer for multiprocessor implementations.

Chapter VI will show how to improve the efficiency of the acts, and most of these improvements will bank on insights obtained by ***program analysis***. Such insights can also help to improve the scheduling rule. It would, for example, be wise to favour (give a higher priority to) any node actor whose activation leads to a ***decrease*** in total store requirements (e.g. queue lengths).

4.8 Actual Implementation

The translation has now been completely described; the remaining chapters merely round the picture off with checking and optimising methods.

The next step would be the actual implementation of the whole matter on a computer. A Lucid system, based on an interpreter, is already available [Ost81, FMY83], and the first compiler passes of that system could be re–used directly for this task. The remaining task would be of the calibre of an M.Sc. project, less than a year's work.

Couldn't we find a simplified version of this translation which would be then easier to implement? First, one would contemplate the omission of *bare* $\boxed{\text{ADVANCE}}$. However, such an implementation would be so hopelessly inefficient as to make the whole exercise pointless. Then, how about omitting $\boxed{\text{NULLIFY}}$ requests? Their importance stems mainly from their vital role in concurrent operators and a simple implementation could do without the latter. One of the main achievements of this thesis has been precisely *not* to rule out concurrency. Omitting concurrency means talking about a much simpler task, disregarding the heart of this thesis. Our protocol is optimised towards concurrency, it looks somewhat clumsy in applications without concurrency. Actually, section 4.6.5 showed ("Trojan Horse") that a $\boxed{\text{COPY}}$ node actor may *have to* produce $\boxed{\text{NULLIFY}}$ requests even if only $\boxed{\text{COMPUTE}}$ and $\boxed{\text{ADVANCE}}$ requests are ever sent to it. The protocol would be incomplete without $\boxed{\text{NULLIFY}}$.

Chapter VI will show how to improve the efficiency of the acts, and most of these improvements will bank on insights obtained by ***program analysis***. Such insights can also help to improve the scheduling rule. It would, for example, be wise to favour (give a higher priority to) any node actor whose activation leads to a ***decrease*** in total store requirements (e.g. queue lengths).

4.8 Actual Implementation

The translation has now been completely described; the remaining chapters merely round the picture off with checking and optimising methods.

The next step would be the actual implementation of the whole matter on a computer. A Lucid system, based on an interpreter, is already available [Ost81, FMY83], and the first compiler passes of that system could be re–used directly for this task. The remaining task would be of the calibre of an M.Sc. project, less than a year's work.

Couldn't we find a simplified version of this translation which would be then easier to implement? First, one would contemplate the omission of ***bare*** $\boxed{\text{ADVANCE}}$. However, such an implementation would be so hopelessly inefficient as to make the whole exercise pointless. Then, how about omitting $\boxed{\text{NULLIFY}}$ requests? Their importance stems mainly from their vital role in concurrent operators and a simple implementation could do without the latter. One of the main achievements of this thesis has been precisely ***not*** to rule out concurrency. Omitting concurrency means talking about a much simpler task, disregarding the heart of this thesis. Our protocol is optimised towards concurrency, it looks somewhat clumsy in applications without concurrency. Actually, section 4.6.5 showed ("Trojan Horse") that a $\boxed{\text{COPY}}$ node actor may ***have to*** produce $\boxed{\text{NULLIFY}}$ requests even if only $\boxed{\text{COMPUTE}}$ and $\boxed{\text{ADVANCE}}$ requests are ever sent to it. The protocol would be incomplete without $\boxed{\text{NULLIFY}}$.

4.9 Closing Remarks

Every language implementation assumes a particular machine as given. The interpreter based Lucid implementation [Ost81, FMY83] simulates a hypothetical ***Lucid machine***, and the interpreter works hard to keep that illusion up. This thesis describes an MPA based Lucid implementation. MPA corresponds closely to the architecture of multi–processors; the main difficulties are in this case hardware specific: how to make the processors communicate, how to allocate actors, how to load the acts. However, it is not very difficult to make even a single physical processor ***appear*** like an array of processors, and this is probably the best way of implementing Lucid until multi–processors become more widely available.

CHAPTER V: Checking the Correctness of the Acts

5.0 Introduction

Much care has been put into the design of the acts, and we have good reason to believe they are mostly correct. This should not keep us from scrutinizing them over and over again. A working implementation would certainly be the most impressive proof of success. But for the time being we rely on *formal* checking methods. The aim of this chapter is to fortify the reader's trust into our design.

If there was a serious flaw in our acts, it would most likely lie in the most complex part of our design, namely the synchronisation of the actors by the protocol. We will therefore design a framework (a **testbed**) in which we can examine the message passing behaviour of actors. We will determine all the **message passing states** for every actor; its message passing behaviour is, at every moment, mostly determined by its current message passing state. The possible state transitions can be summed up in **state transition tables**; this will be illustrated by various examples. The state transitions of a UDF, or *any* net of actors, can be elaborated from the transitions of its components. **Execution logs** are of great help in modelling the actions of an actor, or of a *net*. This will finally be demonstrated by modelling the entire execution of the Sieve program.

As regards difficulty, a big difference must be made between the COPY act and all other acts (↑ beginning of 4.5). The COPY act is a great deal more complicated than all the other node acts, which makes checking the COPY act the most demanding part of this chapter. We will see that even a rather simple COPY (viz. the twin outported one) has an impressive number of states. This is why we have to continually look out for *simplifications* which keep the number of states low; without them, matters are in danger of becoming unmanageable. The correctness of the other acts is by comparison quite obvious, and we discuss them briefly before the COPY act.

5.1 The Testbed

In preparation for our discussion, let us introduce some terms which will play an important role throughout this chapter. We intend to check the correct behaviour of a node actor, and we achieve this by ***placing the actor in a testbed (environment)*** which will confront the actor with all the situations permitted within the protocol (such as all possible sequences of requests and replies, see also [Fau82]).

Let ● be the node actor under examination. Each outport of ● is individually connected to a **demander** (labelled **g**, like "***greedy***"), and each inport of ● is connected to the pertaining **supplier** (labelled **p**, like "***parameter***"). This entire setup (**g** and ● and **p**) is called a **testbed** for the actor ●. The following Lucid graph represents a testbed:

```
                  (outport                 (inport
                    state )                  state)
+----------------+  +------------------+  +----------------+
| demander g     |<-+ node actor ●     |<-+ supplier p     |
+----------------+  +------------------+  +----------------+
```

The actor ● is, of course, in a ***state*** at every moment, and we shall see that some kind of sub-state (a message passing state) can be ascribed to each inport and to each outport. In our implementation, there is ***no queuing*** on the arcs (all the queuing takes place in the $\boxed{\text{COPY}}$ node actors) and the ports at both ends of an arc have thus the same state. The state of a demander or supplier is exactly the state of its ***port***. The state of the actor ● and the state of the testbed are therefore one and the same.

When talking about the message passing at an arbitrary actor port, we will go on using the terminology of ***superiors*** and ***inferiors***. (The arrows in Lucid graphs point always from inferiors to superiors.) In the testbed, **g** can be superior and ● inferior, or ● can be superior and **p** inferior.

```
+----------------+     +----------------+
| superior s     |<----+ inferior i     |
+----------------+     +----------------+
```

5.2 Program Analysis

A proper mathematical proof of correctness (termination and partial correctness) would require us to analyse every act in depth, instruction by instruction. That task alone would double the size, and exceed the aims, of this thesis. Such a proof would certainly be meritable, but it has to be left to the future.

However, some techniques can be readily taken over from proofing, such as ***invariants*** and ***loop termination conditions***. We can indicate only the general approach (i.e. detailed rules will not be given); matters vary greatly among the acts. Most acts are utterly simple, which means there is very little to be analysed.

Most loops in our node acts are ***eternal***, i.e. altogether without termination. Often, no memory is retained from one loop pass to the next, so there are no loop invariants to worry about. Almost every actor **●** is a ***mediator*** between its demander **g** and its supplier **p**: usually, any message from **g** is ***propagated*** in some form to **p**, or vice versa. This message is either a ***request*** (message flow: **g** → **●** → **p**) or a ***reply*** (**p** → **●** → **g**). One can analyse how **●** transforms the message; one should check, in particular, that invariants are not violated. For example, when **●** receives a request, the same daton index must be re–used in the propagated request (while some nodes introduce a fixed index offset); this is all very node sensitive

5.3 Message Passing Behaviour

In another check, we treat the actor like a ***black box***, and examine merely what goes on at its inports and outports, its **message passing behaviour**. If the black box behaves incorrectly, though, one has to take the lid off and put matters right.

The message passing behaviour of an actor can be described by a ***state transition table***, and such a table can reveal where the actor violates design criteria. Let us first look more closely at state transitions, and then recast the protocol into a form convenient for state transition tables.

5.3.1 Message Passing State, and State Transitions

For an actor, each action can be viewed as a state transition, and all the permitted state transitions can be presented as a table (a ***relation*** maps the states to their permitted successor states). Such a table is very useful:

- it reveals the actions which the actor can perform,
- it permits a study of ***concurrent*** actions,
- it enables us to check whether inports and outports adhere to the request protocol,
- it can be used to ***exercise*** a given implementation of the actor. The implementation is correct if the actor can execute each of the listed transitions, and if it never steps outside the alternatives listed.

Such a table can be produced for ***any*** act; we will give examples for some node actors. We will see that the rule for the table generation corresponds closely to the act, both are similar pieces of code. Transition by transition, each table entry (the ***intended*** behaviour) can be compared with the ***true*** behaviour of the actor. This reveals unwanted state transitions in faulty acts. It would even be possible to do some of these checks automatically.

State transitions can be ***non-deterministic***, i.e. an actor can sometimes choose between a number of next states. Furthermore, there is always the extra choice of carrying out only ***part*** of what is possible, or of even doing nothing (successor state being equal to the present state). Such transitions have obviously a ***delaying*** effect. The act design is such that the overall computation result (of the Lucid program) is deterministic even though the execution may be non-deterministic.

Since we are only trying to model the message passing behaviour, we can often ignore those parts of the actor state which have no direct effect on that behaviour. We call the resulting state the **message passing state** (which is a ***function*** of the total state of the actor). As far as message passing is concerned, the choice of successor state is narrowed down by:

(1) the present message passing state of the actor **e**,

(2) the action of the demander(s) **g**,

(3) the action of the supplier(s) **p**.

The message passing state of an actor is made up of the states of its outports, possibly an internal state, and the states of its inports. Different formats are used for the message passing state of the various node types; there is no universal pattern suitable for all actors. There is one general rule: in all message passing states, the state of each inport or outport is always expressed through a ***message label*** (↑ 5.3.2). An example message passing state is (explained in 5.5.1):

D1. .2. A

5.3.2 Protocol Execution and Message Labels

In section 4.2 we have agreed on a universal protocol. Every node actor port is at every moment in a particular state of protocol execution (a ***port state***), and the protocol permits only select successor states. The **port state** is determined by the ***last message which traversed the port***. The message passing partners have no "knowledge" of the internal state of one another. It is therefore appropriate to denote their states in a format which gives the ***port states*** particular prominence. If two ports are ***connected by an arc*** their states are unavoidably ***identical***. We abridge each port state into a single character, called a **message label**, according to:

N	NULLIFY	(request, flowing upstream	**g** → **e** → **p**)
C	COMPUTE	"	" "
A	ADVANCE	"	" "
K	ADVANCE, finalindex	"	" "
D	DATON	(reply, flowing downstream	**p** → **e** → **g**)

state of the actor). As far as message passing is concerned, the choice of successor state is narrowed down by:

(1) the present message passing state of the actor **e**,

(2) the action of the demander(s) **g**,

(3) the action of the supplier(s) **p**.

The message passing state of an actor is made up of the states of its outports, possibly an internal state, and the states of its inports. Different formats are used for the message passing state of the various node types; there is no universal pattern suitable for all actors. There is one general rule: in all message passing states, the state of each inport or outport is always expressed through a ***message label*** (↑ 5.3.2). An example message passing state is (explained in 5.5.1):

D1. .2. A

5.3.2 Protocol Execution and Message Labels

In section 4.2 we have agreed on a universal protocol. Every node actor port is at every moment in a particular state of protocol execution (a ***port state***), and the protocol permits only select successor states. The **port state** is determined by the ***last message which traversed the port***. The message passing partners have no "knowledge" of the internal state of one another. It is therefore appropriate to denote their states in a format which gives the ***port states*** particular prominence. If two ports are ***connected by an arc*** their states are unavoidably ***identical***. We abridge each port state into a single character, called a **message label**, according to:

N	NULLIFY	(request, flowing upstream **g** → **e** → **p**)
C	COMPUTE	" " "
A	ADVANCE	" " "
K	ADVANCE, finalindex	" " "
D	DATON	(reply, flowing downstream **p** → **e** → **g**)

We have to keep the number of port states low since the state tables would otherwise become unmanageable. **N** doubles up as the universal indicator for "the inferior is dormant", and it is thus the initial state (whenever an actor is dormant we pretend that it has just received a $\boxed{\text{NULLIFY}}$ request **N**). **C** doubles up as indicator for "a $\boxed{\text{COMPUTE}}$ request has just been sent". These two states (**C** and **N**) are special in that the inferior can leave them only with the cooperation and initiative of its superior. The protocol boils down to:

message label	next possible action:
.	(we will always print **N** as "."),
N	the superior can change it to **C**, **A** or **K**,
C	the superior can change it to **N** or
C	the inferior can change it to **D** (whichever is first),
A	the inferior can change it to **N**,
K	no change possible,
D	the superior can change it to **N**.

Explanation: if the protocol execution has reached the point where the inferior is ***dormant*** (**N**), it is the superior's turn to issue a **C**, **A** or **K** request; without this, nothing can happen. If the superior requests **A**, the inferior accepts it, and becomes dormant. The latter action is expressed in a state change to **N**. The inferior takes also further appropriate measures, of course, but they are invisible as we concentrate on the messages traversing the port. The message passing reaches its ***terminal state*** once the superior issues a **K** request; no further message will ever go through that port.

On the other hand, after the superior has issued a **C** request, the superior is free to ***nullify*** (**N**) that request again; alternatively, the superior can wait until the inferior is ready to deliver the daton value (**D**). There is even a third possibility: even while the inferior is ready to deliver the daton value, the superior is free to ***delay*** as long as it likes before it decides either for **D** or **N** (such **delay transitions** will usually not be shown in our tables).

The last paragraph glanced over an important point by making a quiet assumption. Whenever a superior nullifies a **C** request, it changes the port state to **N**, and this means that the inferior is now in the dormant state. But the superior can hardly ***force*** its inferior straight from **C** into the dormant state. Instead, the inferior must first accept the $\boxed{\text{NULLIFY}}$ request **N**, take appropriate action (which might include request propagation), and it goes dormant only then. We would have to extend our acts slightly if we wanted them to handle this revised protocol. On the other hand, this simplified protocol has its advantages: avoidable states are a real nuisance in our later discussion, and the simplified protocol is very ***efficient*** in execution. We will not detail the changes which have to be made either to the code, or to our modelling of the message passing.

We will print the protocol state **N** in our tables always as **full stop** ("**.**"); we use this character generally for states of the nature "***nothing special to report***". Tables are easier to read this way: unusual states become much more conspicuous.

5.3.3 Execution in Ultra Priority

The scheduler (↑ 4.7) gives to actors in ***ultra*** priority pre-emption over the ones in ***normal*** priority. Each act lays down which actions take place in which priority. (The exception handling code is executed strictly in ultra priority, and the acts must be of such design that the ***expensive*** proper computations are not carried out in ultra priority.) The ***testbed*** is in normal priority as long as none of the participants is ready to do any exception action. For fundamental operators this reads: execution is in normal priority as long as

- the outport state is not **A** or **K**, and
- no inport state is **A**, and

– the outport state is not "." while an inport state is **C**.

The formula is more complicated for $\boxed{\text{COPY}}$ node actors (↑ 5.5.2).

In the following description, we assume as given a global variable $\boxed{\text{normalEx}}$ which is $\boxed{\text{TRUE}}$ only during execution in normal priority. (In the state transition tables, below, states whose transitions take place in *ultra* priority are marked ↯.)

At the first reading, you may pretend execution were always in *normal* priority. The *ultra* mechanism is meant only to inhibit wasteful state transitions, and it had to be mentioned here because we will refer to it in the following.

5.3.4 Actions of a Demander

There is one demander (g) per outport of actor ●. A demander is only able to inspect and change the respective *outport state* of ●. It can issue **C**, **A** or **K** requests if that outport state is **N**, it can revoke **C** requests (change that outport state from **C** to **N**), or it can accept daton values (change from **D** to **N**).

```
begin
  OS := the respective outport state ;
  ML := the message label in OS ,

  if   ML = N       (i.e. this outport dormant)
  then begin
         the message label in OS may be changed to A , or
         the message label in OS may be changed to K , or
       if  normalEx  then
         the message label in OS may be changed to C ;
       end ;

  if      ML = C    or    (ML = D    and    normalEx)
  then    the message label in OS may be changed to N ,
end ;
```

5.3.5 Actions of a Supplier

There is one supplier (p) per inport of actor ●. The supplier accepts any request; as response, it can merely inspect and change the respective *inport state* of ●. – The supplier acknowledges **A** requests by changing the inport state to **N**, whereas there is *no* acknowledging action for **K** nor for **N** requests. If the inport state is **C** (i.e. after a **C** request), the supplier can respond with sending a daton value (**D**) as reply.

```
begin
  ML :=  the respective inport state ;

  if      ML = A
  then    the inport state may be changed to N ;

  if    ( ML = C )    and    normalEx
  then    the inport state may be changed to D ;
end ;
```

5.4 Checking Node Actors other than COPY

It is not difficult to check that the node acts (↑ 4.5) conform to the *protocol*. Most node actors *propagate* each request and reply via their own opposite ports, possibly with changes to the message content but rarely with a changed message *type*. Such actors will leave everything intact provided the original requests and replies are given correctly. — A mere glance shows that the actors for WRITE, READ and constant (which have only *one* communication partner) generate correct requests or replies, respectively.

After the actor • has received a NULLIFY or ADVANCE request, it takes the appropriate measures and becomes eventually dormant; similar action is taken after each delivery of a daton value. All this is in sympathy with the protocol. Most of our actors become dormant even after ADVANCE, finalindex, but in doing so they are only "overfulfilling" their task, which has no bad consequence.

Our simplification of the protocol permits issuing a new request right after a NULLIFY, even *before* the inferior has reacted upon the NULLIFY. Our acts would not handle this (but can be modified to handle it), but require the superior to be *held up* (delayed) until the inferior has taken the necessary steps.

Most acts use the procedure $\boxed{\text{GetDaton}}$ for the acquisition of daton values. $\boxed{\text{GetDaton}}$ implements the rule that **C** can only be followed by **N** (from the superior) or **D** (from the inferior), and it is not hard to see that $\boxed{\text{GetDaton}}$ performs this task correctly.

Little new can be said about $\boxed{\text{Act\_Root}}$ and the UDF acts. The Lucid graph and the net of actors are related through a ***bijection***, and incorrectness could only be due to an error in the ***translation*** (but the translator program has been carefully tested, ↑ appendix C). — The UDF subnet creation is transparent to requests (except for initial $\boxed{\text{ADVANCE}}$ exceptions), and the UDF actor enters eventually the procedure $\boxed{\text{Pass\_Through}}$. That procedure was designed to be transparent to all messages (i.e. all messages are passed on without change), and it is easily inspected for correctness. — Every UDF subnet is composed of fundamental operators and again of UDFs, and the correctness of the UDF depends on the correctness of these ***constituents***, of course.

Example ($\boxed{\text{FBY}}$ actor)

The message passing state of a $\boxed{\text{FBY}}$ actor can be characterised by:

```
<g, p, P>  where
 g          =  a message label representing the outport state
    p       =  a message label for the state of the left inport
       P    =  a message label  "   "    "   "   " right inport
<., ., .>   is the initial state.
```

We write the states throughout in an order such that outports are on the ***left*** and inports on the ***right***; requests flow therefore left–to–right, replies flow right–to–left. In our tables, the message passing states are written without the angle brackets and without the commas, and the identity transitions (i.e. delay, no change) are not shown at all.

The state transitions of $\boxed{\text{FBY}}$ are then:

no	state		no	state	why	no	state	why	no	state	why
0	. . .		3	C. .	g	7	A. .	g	10	K. .	g
1	.C.	u	0	. . .	e	8	AC.	g	11	KC.	g
"						7	A. .	ge	10	K. .	ge
2	.D.	u	0	. . .	e	9	AD.	g	12	KD.	g
"						7	A. .	ge	10	K. .	ge
3	C. .		0	. . .	g	4	CC.	e	1	.C.	ge
4	CC.		1	.C.	g	5	CD.	p	2	.D.	gp
5	CD.		6	D. .	e	2	.D.	g			
6	D. .		0	. . .	g						
7	A. .	u	13	.K.	e						
8	AC.	u	7	A. .	e						
9	AD.	u	13	.K.	e						
10	K. .	u	29	KKK	e						
11	KC.	u	10	K. .	e						
12	KD.	u	29	KKK	e						
13	.K.		17	CK.	g	21	AK.	g	25	KK.	g
14	.KC	u	13	.K.	e	22	AKC	g	26	KKC	g
"						21	AK.	ge	25	KK.	ge
15	.KD	u	13	.K.	e	23	AKD	g	27	KKD	g
"						21	AK.	ge	25	KK.	ge
"			but not:			19	CKD	g	17	CK.	ge
16	.KA	u	13	.K.	P	24	AKA	g	28	KKA	g
"						21	AK.	gP	25	KK.	gP
17	CK.		13	.K.	g	18	CKC	e	14	.KC	ge
18	CKC		14	.KC	g	19	CKD	P	15	.KD	gP
19	CKD		20	DK.	e	15	.KD	g			
20	DK.		13	.K.	g						
21	AK.	u	16	.KA	e						
22	AKC	u	21	AK.	e						
23	AKD	u	16	.KA	e						
24	AKA	u	21	AK.	P						
25	KK.	u	29	KKK	e						
26	KKC	u	25	KK.	e						
27	KKD	u	29	KKK	e						
28	KKA	u	25	KK.	P						
29	KKK		nothing								

30 states (out of 125), 63 transitions

The table lists all the states which can be reached from the initial state. The states are numbered (**no**) from 0 to 29, with the initial state at number 0, and with the terminal state at the end. The possible successor states are listed on the right of the } sign. If there is a **u** to the left of the } sign, it indicates execution being in *ultra* priority. No further state change is possible once all input states have become **K** (i.e. ***identity*** is the only transition possible). For convenience, both the successor state and its number are given, and in the **why** column some letters (**g**, **e**, **p**, **P**) indicate which actor was the cause for the transition (**g** = demander, **e** = FBY actor, **p** = left supplier, **P** = right supplier). The ***upper*** half of the FBY table are those states where the left FBY operand **p** is still under consideration; in the ***lower*** half all datons come from the right operand **P**.

Example (constants, READ, Identity Operator)

Here is another example, the state transitions of the constant or the READ actor:

no	state		no	state	why	no	state	why	no	state	why
0	.		1	C	g	3	A	g	4	K	g
1	C		2	D	e						
2	D		0	.	g						
3	A	u	0	.	e						
4	K		nothing								

5 states (out of 5), 6 transitions.

This table has moreover a second use: if we take an ***empty*** testbed and connect the supplier ***directly*** to the demander (i.e. merely with an arc in between), the table would describe the behaviour of the resulting system (substitute **p** for **e**).

Example (WRITE)

The table is even simpler for WRITE:

no	state		no	state	why
0	.		1	C	e
1	C		2	D	p
2	D		3	A	e
3	A	u	0	.	p

Example (concurrent binary pointwise operator)

The third example are the state transitions of a concurrent binary pointwise commutative operator, such as ***concurrent** PLUS*. The behaviour of concurrent operators like $\boxed{\text{OR}}$ is more difficult to model, since their choice of transition is ***data sensitive***; they have a few more states, as indicated in the table.

In this example, the states can be written in the same format as in the $\boxed{\text{FBY}}$ example above. We can, however, take advantage of the commutativity, which means that the suppliers **p** and **P** are ***interchangeable***. Many states of our actor come in pairs, where each state results from the other by swapping the inports; our table contains an entry only for either (it is immaterial by which rule we choose either state). Whenever a transition leads to the ***swapped*** counterpart of a state **x**, we indicate this in our table by priming (**x'**) the state number. There are even cases where both state **x** and state **x'** are among the possible outcomes of transitions. In such cases the state number is printed with a double prime (**x''**), with the **why** field telling only either story.

Here is the table of state transitions:

no	state		no	state	why	no	state	why	no	state	why	
0	...	}	8	C..	g	16	A..	g	25	K..	g	
1	..C	u}	0	...	e	16	A..	ge	25	K..	ge	
"		}				17	A.C	g	26	K.C	g	
2	..D	u}	0	...	e	16	A..	ge	25	K..	ge	
"		}				18	A.D	g	27	K.D	g	
3	..A	u}	0	...	p	16	A..	ep	25	K..	ep	
"		}				19	A.A	g	28	K.A	g	
4	.CC	u}	0	...	e	16	A..	ge	25	K..	ge	
"		}				21	ACC	g	30	KCC	g	
5	.CD	u}	0	...	e	16	A..	ge	25	K..	ge	
"		}				22	ACD	g	31	KCD	g	
6	.DD	u}	0	...	e	16	A..	ge	25	K..	ge	
"		}				23	ADD	g	32	KDD	g	
7	.AA	u}	0	...	pP	16	A..	gpP	25	K..	gpP	
"		}	3"	..A	p	19"	A.A	gp	28"	K.A	gp	
"		}				20	AAA	g	34	KAA	g	
8	C..	}	0	...	g	11	CCC	e	4	.CC	ge	
9	C.C	}	1	..C	g	10	C.D	P	2	..D	gP	
10	C.D	}	14	D..	e	2	..D	g				
11	CCC	}	4	.CC	g	5"	.CD	gP	6	.DD	gpP	
"		}				12"	CCD	gP	13	CDD	gpP	
12	CCD	}	5	.CD	g	13	CDD	p	6	.DD	gp	
"		}				10'	CD.	ep	2'	.D.	gep	
"		}				9'	CC.	e	1'	.C.	ge	
"		}				14	D..	e	15'	DD.	ep	OR/AND only
13	CDD	}	6	.DD	g	14	D..	e				
14	D..	}	0	...	g							
15	D.D	}	2	..D	g	14	D..	e	0	...	ge	OR/AND only
16	A..	u}	7	.AA	e							
17	A.C	u}	16	A..	e							
18	A.D	u}	7	.AA	e							
19	A.A	u}	16	A..	P							
20	AAA	u}	19"	A.A	p	16	A..	pP				
21	ACC	u}	16	A..	e							
22	ACD	u}	16	A..	e							
23	ADD	u}	7	.AA	e							
24	ADA	u}	19	A.A	e	18'	AD.	P	16	A..	eP	
25	K..	u}	36	KKK	e							
26	K.C	u}	29'	KK.	e							
27	K.D	u}	36	KKK	e							
28	K.A	u}	35'	KKA	e	25	K..	P	29'	KK.	eP	
29	K.K	u}	36	KKK	e							
30	KCC	u}	25	K..	e							
31	KCD	u}	29	K.K	e	32	KDD	p	33	KDK	ep	
32	KDD	u}	36	KKK	e							
33	KDK	u}	36	KKK	e							
34	KAA	u}	28"	K.A	p	25	K..	pP				
35	KAK	u}	29	K.K	p							
36	KKK	}	nothing									

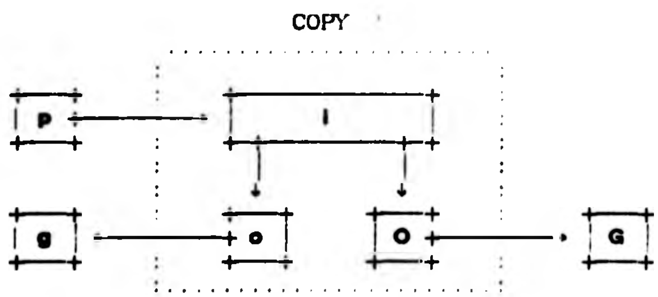
37 states (out of 125), 97 transitions.

The state transition tables so far were all prepared ***by hand***, and though the greatest care has been taken they may contain a slip or two. The examples were meant mainly to illustrate how to describe the behaviour of an actor by a table. The state transition tables for the COPY node actors (↑ 5.5.4 f) are generated by program, and high expectations for their correctness are justified.

5.5 Checking the COPY Node Actors

Modelling the message passing behaviour of the COPY node actors, and thus showing the correctness of the COPY acts, is more difficult.

In our checking of COPY, we re-use the terminology of ***queues***, ***ql*** and novalues (↑ 4.6.4 and 4.6.6). We continue having ***separate actors*** for the COPY inport and outports, but we leave open how COPY manages the ***buffers***. When modelling the behaviour of the COPY node actor and its environment, we are dealing with the participants shown in the following Lucid graph (here: twin outport COPY):



The COPY inport (**I**) is in communication with its supplier (**p** like "***parameter***"), and each COPY outport (**o**) is in communication with its specific demander (**g** like "***greedy***"). In order to differentiate both outports, we label the left side with ***lower*** case letters and the right with ***upper*** case.

5.5.1 Message Passing States of $\boxed{\text{COPY}}$ Node Actors

The universal $\boxed{\text{COPY}}$ act has an arbitrary number of outports, specified only in the $\boxed{\text{COPY}}$ node actor ***creation***. The daton queue in each $\boxed{\text{COPY}}$ outport can hold an ***arbitrary*** number of datons (of type $\boxed{\text{ANYTYPE}}$). We shall now try to condense the state of the $\boxed{\text{COPY}}$ node actor into a manageable form.

For our modelling, it is sufficient to characterise the state of each $\boxed{\text{COPY}}$ outport actor by a triplet:

$\langle m, q, v\rangle$ where
m is a message label (an **outport state**),
q is the daton queue at this outport,
$|q|$ is the number of datons queued at this outport ("*ql*"),
($|q|$, v are non-negative integers)
v is the "novalues" of this outport, such that
$|q| \cdot v = 0$.

Such a $\boxed{\text{COPY}}$ **outport actor state** is clearly not one of the ***outport states*** (↑ 5.3.2); a $\boxed{\text{COPY}}$ outport actor consists of more things than just an outport. This clash of terms is regrettable, but one can live with it.

The initial state of every $\boxed{\text{COPY}}$ outport actor is $\langle N, \mathit{bottom}, 0\rangle$. The state of the $\boxed{\text{COPY}}$ inport actor is just a message label, and it is initially **N**. The state of a complete $\boxed{\text{COPY}}$ node actor is the sequence of the states of its outport actors and of its inport:

$\langle\langle m, q, v\rangle, \quad \iota\rangle$ state of the single-outport COPY,
$\langle\langle m, q, v\rangle, \langle k, p, w\rangle, \iota\rangle$ state of the twin-outport COPY,
$\langle o_0, o_1, \ldots o_{n-1}, \iota\rangle$ in general (n = number of outports).

We intend to model only the message passing behaviour, and we can therefore go one step further. We need to incorporate merely the ***queue lengths*** in the outport actor states, instead of the queues themselves. More precisely, this modified $\boxed{\text{COPY}}$ state is then its **message passing state** (but we omit the words "***message passing***" most of the time). In our tables, we will print the message passing states of $\boxed{\text{COPY}}$ in the order shown above, albeit again without the commas and the angle brackets. An example of a $\boxed{\text{COPY}}$ message passing state is (twin outport $\boxed{\text{COPY}}$):

D1. .2. A

Here, the left $\boxed{\text{COPY}}$ outport actor has one daton queued; it has just delivered the value of that daton, and its demander still has to confirm the acceptance. The right $\boxed{\text{COPY}}$ outport actor has two datons queued, but it is otherwise inactive. The $\boxed{\text{COPY}}$ inport has just issued an **A** request.

An **intermediary state** of the $\boxed{\text{COPY}}$ node actor is any state in which this actor is enabled for further state changes without requiring a state change in any demander. A theorem can be formulated: if k is the minimum of the $\boxed{\text{novalues}}$ of all outports of a $\boxed{\text{COPY}}$ node actor C, then k can be non–zero only in ***intermediary*** states of C.

5.5.2 The Actions of the Participants

How many actors take part in our modelling of $\boxed{\text{COPY}}$, and what is each of them allowed to do? We call an **agent** any actor which might change the $\boxed{\text{COPY}}$ state. As stated before, we have four kinds of agents: the demanders, the outport actors, the inport actor, and the supplier. If n is the number of $\boxed{\text{COPY}}$ outports, there are altogether $2 * (n+1)$ agents. In every state of $\boxed{\text{COPY}}$ at least one agent is enabled for a state change, unless there is a deadlock. Indeed, any number of agents may be enabled for any number of state changes. Each agent carries out at most ***one*** transition in a single go, but ***different*** agents are permitted to "fire" simultaneously.

The rule for $\boxed{\text{normalEx}}$ (* 4.7 and 5.3.1) must be slightly extended. Our model for the $\boxed{\text{COPY}}$ node actor is in ***normal*** priority if simultaneously:

- none of the message labels (inport or outport) is **A**,
- the $\boxed{\text{novalues}}$ is $\boxed{\text{finalindex}}$ at each $\boxed{\text{COPY}}$ outport whose message label is **K**,
- the $\boxed{\text{COPY}}$ inport state is **C** only if at least one $\boxed{\text{COPY}}$ outport is currently interested in the daton value to come.

Section 5.3.4 f stated already the actions of the demanders and of the supplier, but it remains to sum up the actions of the $\boxed{\text{COPY}}$ inport actor and of the $\boxed{\text{COPY}}$ outport actors. These code fragments are closely related to the core of the $\boxed{\text{COPY}}$ act, and they were used almost directly to produce the state tables.

5.5.2.1 Action by the $\boxed{\text{COPY}}$ Inport

The $\boxed{\text{COPY}}$ inport actor (I) is capable of examining and changing any part of the $\boxed{\text{COPY}}$ state (though it would not alter any *outport* message label). The inport actor does all those tasks which concern more than one outport, and it communicates with the outports mainly through the $\boxed{\text{novalues}}$ and the queues. The actions of the inport are essentially:

```
begin
  I      :=  the inport state, being a message label ;

  acond :=  true if all the "novalues" are non-zero ;

  ccond :=  true if for at least one outport:
              the message label is C and
              no daton is queued for that outport ;

  if   ( I = C )
       and
       (acond   or   not ccond)
  then   the inport state may be changed to N ;


  if   the novalues in all outport actor states are finalindex
  then begin
         if    ( I <> C )   and   ( I <> K )  and  acond
         then   the inport state may be changed to K ;
       end

  else begin
         if    I = N
         then begin
                if   acond
                then the inport state may be changed to A
                     but then also
                     must each novalues be decremented by one

                else if    ccond   and   normalEx
                     then  the inport state may be changed to C ;
              end ;

         if    I = D
         then begin     (one may do the following, all in one go.)
                for each outport
                do  if     its novalues is greater zero
                    then  reduce its novalues by one,
                    else  append the daton to its daton queue ;

                but then also
                set the inport state to A ;
end    end    end ;
```

5.5.2.2 Action by a $\boxed{\text{COPY}}$ Outport

There is one $\boxed{\text{COPY}}$ outport actor for each $\boxed{\text{COPY}}$ outport. An outport actor (o) is capable of examining and changing any part of the $\boxed{\text{COPY}}$ state (though the only message label it would alter is its *own* one). Each outport cooperates, of course, closely with the inport. The actions of an outport are essentially:

```
begin
  OS := the respective outport actor state ;
  ML := the message label in OS ;

  if    ML = C      and  normalEx
        and   a daton is queued at this outport
  then  the message label in OS may be changed to D ;

  if    ML = K
  then  the novalues in OS may be changed to finalindex ,

  if    ML = A
  then begin     (one may do the following, all in one go:)
         if    a daton is queued at this outport
         then  pop the oldest daton off that queue
         else  increment by one the novalues in OS ;

         but then also
         change the message label in OS to N ;
       end ;
end ;
```

5.5.3 Simplifications

The message passing state of the $\boxed{\text{COPY}}$ node actor has been presented above, and it was obtained by pruning the total state of $\boxed{\text{COPY}}$. Before we generate the state transition table of a $\boxed{\text{COPY}}$ node actor, we apply the following simplifications to bound and reduce the number of states.

1) We pretend that the $\boxed{\text{COPY}}$ node actor memorises only the ***difference*** between inport index and outport index. It is easy to see that the actor handles the ***absolute*** inport index correctly.

2) We ignore the detailed contents of the daton queues; after all, even the $\boxed{\text{COPY}}$ node actor does not analyse the daton values. We trust that the actor makes no mistake in appending every new daton at the tail of the queue, and popping datons off the head of the queue. We memorise the ***length*** of the daton queue.

closely with the inport. The actions of an outport are essentially:

```
begin
  OS := the respective outport actor state ;
  ML := the message label in OS ;

  if    ML = C      and   normalEx
        and   a daton is queued at this outport
  then  the message label in OS may be changed to D ,

  if    ML = K
  then  the novalues in OS may be changed to finalindex ;

  if    ML = A
  then begin     (one may do the following, all in one go:)
         if    a daton is queued at this outport
         then  pop the oldest daton off that queue
         else  increment by one the novalues in OS ;

         but then also
         change the message label in OS to N ;
       end ;
end ;
```

5.5.3 Simplifications

The message passing state of the COPY node actor has been presented above, and it was obtained by pruning the total state of COPY. Before we generate the state transition table of a COPY node actor, we apply the following simplifications to bound and reduce the number of states.

1) We pretend that the COPY node actor memorises only the ***difference*** between inport index and outport index. It is easy to see that the actor handles the ***absolute*** inport index correctly.

2) We ignore the detailed contents of the daton queues; after all, even the COPY node actor does not analyse the daton values. We trust that the actor makes no mistake in appending every new daton at the tail of the queue, and popping datons off the head of the queue. We memorise the ***length*** of the daton queue.

3) Our $\boxed{\text{COPY}}$ node actor is ***demand driven***; only the arrival of a request (at an outport), or the arrival of a daton value (at the inport), can cause a state transition. We omit in our tables extra states which are due to delays inside $\boxed{\text{COPY}}$. We assume instead "if $\boxed{\text{COPY}}$ can act, it will".

4) We shall study only the $\boxed{\text{COPY}}$ node actor with ***one*** outport, and the $\boxed{\text{COPY}}$ node actor with ***two*** outports. Any $\boxed{\text{COPY}}$ with more outports can be built from the latter.

In the outport actor state, ***queue length*** = 0 and $\boxed{\text{novalues}}$ = 0 are both printed as dot ".", and $\boxed{\text{novalues}}$ = $\boxed{\text{finalindex}}$ prints like $\boxed{\text{novalues}}$ = 1.

5.5.4 Single outport $\boxed{\text{COPY}}$

We study first the single-outport $\boxed{\text{COPY}}$ node actor. Such a $\boxed{\text{COPY}}$ node actor can at best have ***one*** daton queued (in pipeline ddDF without bulk requests), and its $\boxed{\text{novalues}}$ is non-zero only in ***intermediary*** states. The state table is therefore reasonably small.

no	state	no	state	why	no	state	why	no	state	why
0	. . . .	1	C . . .	g	2	A . . .	g	3	K . . .	g
1	C . . .	0	. . . .	g	4	. . . C	gi	5	C . . C	i
2	A . . . u	19	. . 1 .	o						
3	K . . . u	21	K . 1 .	o	24	K . 1 K	oi			
4	. . . C u	0	. . . .	i						
5	C . . C	4	. . . C	g	6	. . . D	gp	7	C . . D	p
6	. . . D	16	. 1 . A	i						
7	C . . D	6	. . . D	g	16	. 1 . A	gi	17	C1 . A	i
"		18	D1 . A	oi						
8	. . . A u	0	. . . .	p						
9	A . . A u	2	A . . .	p	19	. . 1 .	op	22	. . 1 A	o
10	K . . A u	3	K . . .	p	21	K . 1 .	op	23	K . 1 A	o
"		24	K . 1 K	oi						
11	. 1 . .	12	C1 . .	g	14	A1 . .	g	15	K1 . .	g
12	C1 . .	11	. 1 . .	g	13	D1 . .	o			
13	D1 . .	11	. 1 . .	g						
14	A1 . . u	0	. . . .	o						
15	K1 . . u	21	K . 1 .	o	24	K . 1 K	oi			
16	. 1 . A u	11	. 1 . .	p						
17	C1 . A u	11	. 1 . .	gp	12	C1 . .	p	16	. 1 . A	g
18	D1 . A u	13	D1 . .	p						
19	. . 1 . u	8	. . . A	i	9	A . . A	gi	10	K . . A	gi
"		20	A . 1 .	g	21	K . 1 .	g			
20	A . 1 . u	9	A . . A	i						
21	K . 1 . u	24	K . 1 K	i						
22	. . 1 A u	19	. . 1 .	p						
23	K . 1 A u	21	K . 1 .	p	24	K . 1 K	i			
24	K . 1 K		nothing							

25 states (out of 75), 50 transitions

The table shows all the states which can be reached from the initial state The states are numbered (**no**) from 0 to 24; their order is due to a hash function (not to be explained here). For further detail refer to the explanations after the FBY table (↑ 5.4). In the **why** column some letters (**g**, **o**, **i**, **p**) indicate which actor caused the transition (**g** = demander, **o** = outport actor, **i** = inport actor, **p** = supplier) The message passing states are written in the format defined in section 5.5.1:

	——— outport actor state.		———	inport actor state:
(components:)	outport state,	queue length,	novalues	inport state
(represent'n.)	message label,	integer,	integer	message label
(example:)	N	0	0	N

The message label N, as well as queue lengths or $\boxed{\text{novalues}}$ of 0, are all printed as dot (for example . . . . . is the initial state NOO N). Let us study, as an example, one line of the table:

```
7 C. .  D  |  6   . . .  D g  | 17  C1.  A i  | 16  .1.  A gi
"          | 18   D1.  A oi   |               |
```

It shows state number 7, which has 4 successors to choose from; execution is in normal priority. At the outset, the queue length and $\boxed{\text{novalues}}$ are both zero, the demander is waiting for a daton from $\boxed{\text{COPY}}$ (it has set the outport message label to C), and the supplier has just delivered a daton to $\boxed{\text{COPY}}$ (setting the inport state to D). Incidentally, there is just one reference to state 7, but other states have up to 5 references (e.g. state 24).

A transition is made to state 6 if the demander nullifies the **C**. State 17 results if the $\boxed{\text{COPY}}$ inport ***queues*** the daton, also issuing an **A** to the supplier (the two go always together). $\boxed{\text{COPY}}$ gets into state 16 if the demander and the inport happen to act (as described) at the ***same*** time. On the other hand, ***immediately*** after the inport has queued the daton, and has issued **A**, the outport may send that daton to the demander, thus setting the outport message label to **D**. This puts the $\boxed{\text{COPY}}$ into state 18. This action by the outport (changing to **D** the message label in the outport actor state) would of course be irreconcilable with a nullification by the demander (as in states 6 or 16, changing that label to **N**). If these opposed intentions collide during program execution, the message passing mechanism will take a ***non***-deterministic choice. Both choices give ultimately the same effect (due to the design of the acts, look at $\boxed{\text{GetDaton}}$), it would have been wrong to resolve this situation by ***priorities***.

5.5.5 Twin outport $\boxed{\text{COPY}}$

The first thing one notices when comparing the twin–outport $\boxed{\text{COPY}}$ node actor to the single–outported one is the far larger number of states. With every new outport the number of states grows by a factor of about 15, since only few ***symmetries*** can be exploited to reduce the table size. The transition table for the twin–outported $\boxed{\text{COPY}}$ node actor has 304 states with altogether 1619 transitions. Because of its size, the complete table has been put into appendix E, we give here only the necessary explanations, and discuss a few example transitions.

Two further simplifications have been employed in the state transition table for the twin–outport $\boxed{\text{COPY}}$:

5) Our state transition table comprises queues only up to a finite maximum length, and we choose this maximum to be ***two***. Our checking method resembles mathematical induction, and this requires one proof for a starting value and one proof for the induction step. To be correct we would have to demonstrate both the ***increase*** and the ***reduction*** of the queue, and we would have to do this both for the ***minimum*** queue and for an ***arbitrary*** queue. However, since queues have a ***linear*** law of growth and shrinkage, we outstretch nobody's trust when demonstrating the growing and shrinking of queues only up to a queue length of two.

6) Similarly, the table comprises $\boxed{\text{novalues}}$ only up to a finite maximum value, and we choose this maximum to be two.

The table lists only current states with queues of a length limited to one, and with its $\boxed{\text{novalues}}$ also limited to one. If one of the successor states has a queue length or a $\boxed{\text{novalues}}$ ***greater one***, the successor state number is followed by a minus ("–").

It is irrelevant which way round the outports are numbered; if two states result from each other by ***permuting*** (swapping) the outports, we can avoid printing table entries for both (it is immaterial by which rule we choose either state). Whenever a

transition leads to the swapped counterpart of a state x, we indicate this in our table by priming (x') the state number. There are even cases where *both* state x and state x' are among the possible outcomes of transitions. In such cases the state number is printed with a double prime (x"), with the why field telling only either story.

For example, here are the transitions for one state:

no	state	no	state	why	no	state	why
1	... C.. .	0	... .	G	4	C.. C.. .	g
"		1'	C..	gG	5'	A.. C.. .	g
"		2'	A..	gG	6'	K.. C.. .	g
"		3'	K..	gG	10	 C	Gi
"		14	C.. C.. C	gi	11"	... C.. C	i
"		15'	A.. C.. C	gi	12'	A.. ... C	gGi
"		16'	K.. C.. C	gi	13'	K.. ... C	gGi

This example for state 1 shows 15 successor states (the double primed state counts double). Together with the *identity* transition, there are 16 (= 4*4) successor states, since either outport can end up in the state **N**, **C**, **A** or **K**.

5.6 Discussion of the State Transition Tables

Foolish States

The transition tables contain a fair number of "foolish" states, i.e. states which appear somehow unreasonable. Look for example at state 70 (A . . C1 . C): the COPY inport requests a daton (from the operand actor **p**) though the daton is not required by either COPY outport. It is the purpose of execution in ultra priority to get actors as quickly as possible out of such foolish states; it minimises also their chance of getting into such a state, in the first place.

Execution Logs

A particular sequence of requests has been discussed in section 4.6.5, where a COPY node actor is forced into requesting the evaluation of a daton even though neither outport has expressed a wish for that daton (we called it ***despair***). The situation was saved when, in the middle of the daton evaluation, an **A** request was received whereupon the evaluation could be nullified. We have in the meantime obtained the tools to express this whole scenario much more clearly: we simply write down the $\boxed{\text{COPY}}$ states in the sequence in which they are encountered. This is the simplest form of an **execution log**, a graphic representation of how a computation progresses. The vertical axis is the ***time*** coordinate.

Example (Despair)

```
COPY state
... ... .  (initial state)
A.. ... .  demander g issues bare A
..1 ... .  COPY resolves A
C.1 ... .  demander g issues C
C.1 ... C  COPY issues C out of despair
C.1 A.. C  demander G issues bare A
C.1 ..1 .  COPY nullifies the C request
C.. ... A  COPY propagates A
C.. ... .  supplier resolves A
C.. ... C  COPY propagates the original C
C.. ... D  supplier delivers D
D1. .1. A  COPY accepts D, and generates A
D1. .1. .  supplier resolves A
A1. .1. .  demander g accepts D, and generates A
... .1. .  COPY resolves A
... A1. .  demander G issues bare A
... ... .  COPY resolves A
```

Note that $\boxed{\text{COPY}}$ issues a $\boxed{\text{NULLIFY}}$ request ("." after **C**) to the supplier although neither of the $\boxed{\text{COPY}}$ outports ***received*** a $\boxed{\text{NULLIFY}}$ request. This shows that this Lucid implementation (with concurrency) would be incomplete without $\boxed{\text{NULLIFY}}$ requests.

Example (Trojan Horse)

The ***Trojan Horse*** situation (↑ 4.6.5) can be expressed with similar ease:

```
COPY state
........ (initial state)
A....... demander g issues A
..1..... COPY resolves A
C.1..... demander g issues C
C.1....C COPY issues C out of despair
C.1....D supplier delivers D
C...1. A COPY accepts D, and generates A
C...1. . supplier resolves A
C...1. C COPY propagates the original C
C...1. D supplier delivers D
D1. .2. A COPY accepts D, and generates A
D1. .2. . supplier resolves A
A1. .2. . demander g accepts D, and generates A
... .2. . COPY resolves A
... A2. . demander G issues bare A
... .1. . COPY resolves A (Trojan Horse is discarded)
... A1. . demander G issues bare A
... ... . COPY resolves A
```

States of UDFs

The state transition table for a *UDF* can be obtained by first generating the cross product of the tables of the components, and by then eliminating ***incoherent*** states. A UDF state is **incoherent** if the UDF contains anywhere an outport state whose message label differs in the inport it is connected to. ***Equivalent*** states (states which cannot be distinguished from outside) have to be eliminated, too. – This method is altogether rather laborious, and we will not deal with it further than this.

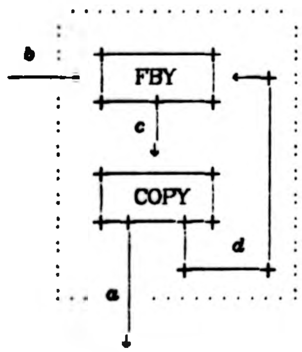
Alternatively, we can place the UDF in a testbed, "play through" all the possible request sequences, and write down the emerging successor states of the UDF.

Example ($\boxed{\text{FIRST}}$ actor)

The UDF $\boxed{\text{FIRST}}$ is defined (in terms of fundamental operators):

```
FIRST (a)  = p  WHERE   p = a FBY p   END ;
```

A $\boxed{\text{FIRST}}$ act can thus be built from $\boxed{\text{FBY}}$ and $\boxed{\text{COPY}}$. The state transition table of $\boxed{\text{FIRST}}$ can be generated from those of $\boxed{\text{FBY}}$ and the twin-outport $\boxed{\text{COPY}}$. For this, we make a table with one row for the state of each actor. We label each arc (with letters *a* ... *d*), and since certain portions in each actor state correspond to that arc (viz. the port states) we can write the appropriate letter also into the message passing state of the actor (we use "?" as placeholders for miscellanies).



act	← states →	
	identities	initially
COPY	*a*?? *d*?? *c*	... .
FBY	*cbd*	...
FIRST	*ab*	..

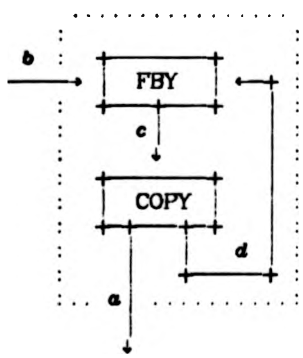
We transpose the table, above, and write successive states on successive lines, so that the vertical axis represents time. In this way we get again an **execution log**, now for a ***system*** of two actors. Let us play through an example where we send **A** and then **C** to the $\boxed{\text{FIRST}}$ actor:

Example ($\boxed{\text{FIRST}}$ actor)

The UDF $\boxed{\text{FIRST}}$ is defined (in terms of fundamental operators):

```
FIRST (a)  = p  WHERE   p = a FBY p   END ;
```

A $\boxed{\text{FIRST}}$ act can thus be built from $\boxed{\text{FBY}}$ and $\boxed{\text{COPY}}$. The state transition table of $\boxed{\text{FIRST}}$ can be generated from those of $\boxed{\text{FBY}}$ and the twin–outport $\boxed{\text{COPY}}$. For this, we make a table with one row for the state of each actor. We label each arc (with letters *a* ... *d*), and since certain portions in each actor state correspond to that arc (viz. the port states) we can write the appropriate letter also into the message passing state of the actor (we use "?" as placeholders for miscellanies).



act	states: identities	states: initially
COPY	*a?? d?? c*	... .
FBY	*cbd*	...
FIRST	*ab*	..

We transpose the table, above, and write successive states on successive lines, so that the vertical axis represents time. In this way we get again an **execution log**, now for a ***system*** of two actors. Let us play through an example where we send **A** and then **C** to the $\boxed{\text{FIRST}}$ actor:

```
        COPY,    FBY   (acts)                              FIRST state
     e?? d?? c,  cbd   (identities)                     (see table below)

(0)  ... ... .   ...   (initial state)                           0
(1)  A.. ... .   ...  demander issues A                          6
     ..1 ... .   ...  COPY resolves A                            0
(2)  C.1 ... .   ...  demander issues 1st C                      3
     C.1 ... C   C..  COPY (desperate) issues C to FBY           3
     C.1 ... C   CC.  FBY  propagates C to FIRST's supplier      4
     C.1 ... C   CD.  supplier delivers D                        5
     C.1 ... D   D..  FBY  passes D back to COPY                 -
     C.. .1. A   A..  COPY accepts D, and generates A            -
     C.. .1. .   .K.  FBY  resolves A, FIRST's supplier "dies"   -
(3)  C.. .1. C   CK.  COPY propagates C to FBY                   -
     C.. C1. C   CKC  FBY  propagates C to right COPY outport    -
     C.. D1. C   CKD  right COPY outport delivers D              -
     C.. .1. D   DK.  FBY  passes back D                         -
     D1. .2. A   AK.  COPY accepts D, and generates A           14
     D1. A2. .   .KA  FBY  propagates A                         14
     D1. .1. .   .K.  COPY resolves A (right outport)           14
     A1. .1. .   .K.  demander accepts D, and generates A       15
     ... .1. .   .K.  COPY resolves A (left outport)            12
     C.. .1. .   .K.  demander issues C                         13
     jump to (3)
```

Every actor starts in its initial state, of course. At the beginning, the demander of $\boxed{\text{FIRST}}$ (being also the superior of the left $\boxed{\text{COPY}}$ outport) is alone able to change state. Moving step by step from this point, we can work out all the other relevant states of $\boxed{\text{FIRST}}$. We end up with a table with a certain amount of redundancy:

- non-deterministic state changes *inside* the UDF are of no interest any longer, since we want to model only the message passing behaviour of the UDF as a whole,
- certain components within the actor states change always *together*, and we can condense this repeated information to the essential minimum. (The corresponding state numbers in the state transition table have been printed on the right of the execution log. We see that some successive steps in the log collapse into merely one step in the table, and some intermediary steps have no counterpart in the table at all.)

By playing through all the possible request sequences we get the following state transition table of FIRST (with reference to the ***identities***, above, the state of FIRST is ***ab***, where ***a*** is the outport state and ***b*** is the inport state):

no	state		no	state	why	no	state	why	no	state	why
0	..		3	C.	g	6	A.	g	9	K.	g
1	.C	u	0	..	e	7	AC	g	10	KC	g
"						6	A.	ge	9	K.	ge
2	.D	u	12	.K	e	8	AD	g	11	KD	g
"						15	AK	ge	16	KK	ge
3	C.		0	..	g	4	CC	e	1	.C	ge
4	CC		1	.C	g	5	CD	p	2	.D	gp
5	CD		14	DK	e	2	.D	g			
6	A.	u	0	..	e						
7	AC	u	6	A.	e						
8	AD	u	12	.K	e						
9	K.	u	16	KK	e						
10	KC	u	9	K.	e						
11	KD	u	16	KK	e						
12	.K		13	CK	g	15	AK	g	16	KK	g
13	CK		14	DK	e	12	.K	g			
14	DK		12	.K	g						
15	AK	u	15	.K	e						
16	KK		nothing								

17 states (out of 25), 33 transitions.

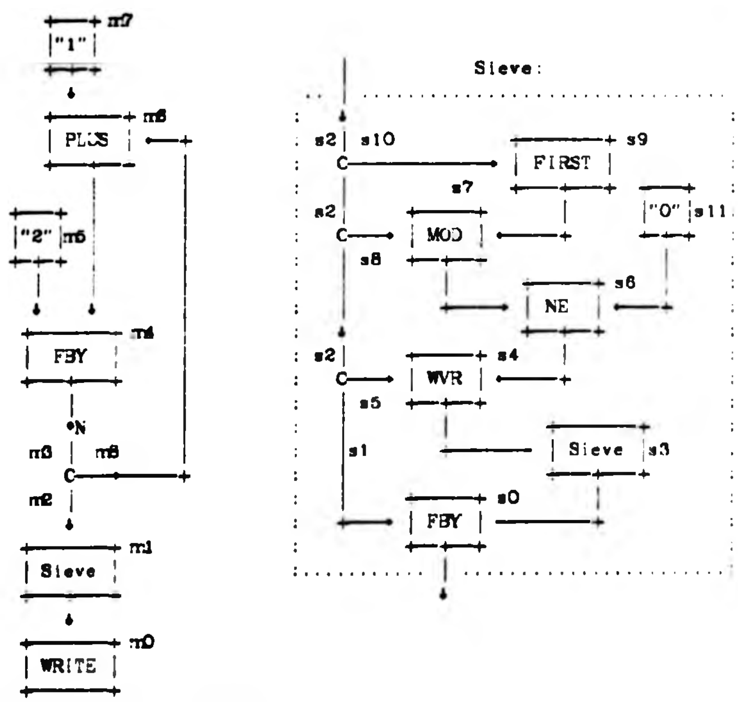
5.7 Example (Sieve): the execution log

Using the Sieve example as illustration, we shall now discuss how the messages pass through the ***net*** of actors, and how this yields the computation result. These actions will be presented in form of the execution logs introduced earlier in this chapter. There will be one log for the main program, and a second log specifically for the Sieve UDF.

The logs represent the state of large composites (e.g. main program and UDF actor) through the states of their ***components***. Earlier in this chapter, the possible state transitions have been listed for most of the components which occur in the Sieve example (MOD, NE, PLUS may be instances of the concurrent binary operator). They have not been described for WVR. WVR is a good deal more complicated than

FIRST, its message passing behaviour depends on its earlier actions (the same is true for the Sieve). We will, nevertheless, write the state of WVR as if it was a ***pointwise*** binary operator (i.e. "***oti***"), and the reader is asked to take its transitions in the log as correct. WVR and FIRST are both UDFs; whatever we learn about the Sieve UDF can benefit our comprehension of these other UDFs. To keep the discussion simple, we treat FIRST and WVR like ***predefined*** operators, non–UDFs.

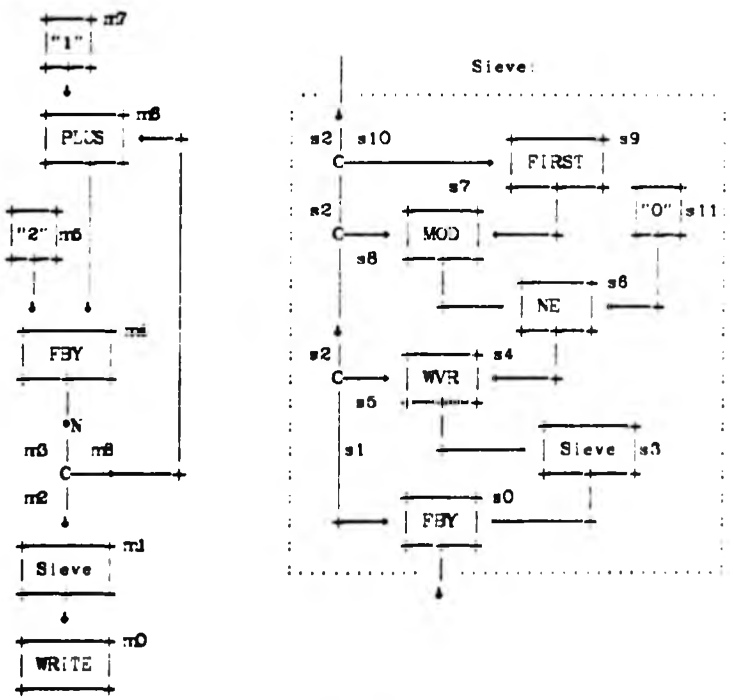
The Sieve state will be written "***oxi***", where ***x*** is either "." or "1", indicating unexpanded or expanded state. Here is the graph of the Sieve program again (↑ 4.3.3.1):



To avoid confusion, the actors in the ***main program*** have their numbers prefixed with an ***m***, while the actors within the Sieve get an ***s***.

FIRST, its message passing behaviour depends on its earlier actions (the same is true for the Sieve). We will, nevertheless, write the state of WVR as if it was a ***pointwise*** binary operator (i.e. "***oti***"), and the reader is asked to take its transitions in the log as correct. WVR and FIRST are both UDFs; whatever we learn about the Sieve UDF can benefit our comprehension of these other UDFs. To keep the discussion simple, we treat FIRST and WVR like ***predefined*** operators, non–UDFs.

The Sieve state will be written "***oxi***", where ***x*** is either "." or "1", indicating unexpanded or expanded state. Here is the graph of the Sieve program again (↑ 4.3.3.1):



To avoid confusion, the actors in the ***main program*** have their numbers prefixed with an **m**, while the actors within the Sieve get an **s**.

Only the actors m0...m8 exist at the very beginning of program execution. They have been created by the root actor (which itself has terminated), and each actor is in its initial state. The $\boxed{\text{Sieve}}$ actor m1 is a UDF actor, and it is yet unexpanded. The state is thus:

acts	actors	identities	initial states
WRITE	m0	*a*	.
Sieve	m1	*a*?*b*	...
COPY	m2,m8,m3	*b*?? *g*?? *c*	... .
FBY	m4	*cde*	...
2	m5	*d*	.
+	m6	*efg*	...
1	m7	*f*	.

The state of the main program is the ***ensemble*** of the states of its components; an example is the entire entry below the heading "initial states".

Initially, all the actors have ***zero priority***, except for the $\boxed{\text{WRITE}}$ actor; its priority is one. $\boxed{\text{WRITE}}$ is therefore the one to take action: it issues a $\boxed{\text{COMPUTE}}$ request (**C**) to the $\boxed{\text{Sieve}}$ m1, and starts waiting for the delivery of a daton value. In doing so, $\boxed{\text{WRITE}}$ becomes suspended, and the actual priority of the $\boxed{\text{Sieve}}$ m1 rises to ***one***. Working out the actual priorities for the remainder of the log might be an interesting exercise.

The $\boxed{\text{Sieve}}$ UDF must be expanded (i.e. the subnet of actors s0...s11 must be created, initialised and bound to the environment) as soon as the attempt is made to send the first $\boxed{\text{COMPUTE}}$ request to the $\boxed{\text{Sieve}}$ m1. This whole process is invisible in the logs. This newly created subnet has things in common with the product of the root actor: all the subnet actors are in their initial states, the $\boxed{\text{Sieve}}$ UDF s3 is yet unexpanded. – Everything that is said about the $\boxed{\text{Sieve}}$ m1 applies correspondingly to $\boxed{\text{Sieve}}$ s3, and to the $\boxed{\text{Sieve}}$ inside s3, etc.

Log of [Sieve] Main Program

The table, above, can also be transposed (and some superfluous detail can be omitted), and the resulting execution log for the [Sieve] *main program* goes as follows:

```
       Sieve   COPY        FBY   PLUS      (acts)
        m1,  m2,m8,m3,     m4,   m6        (actors)
        a?b, b?? g?? c,  cde,  efg        (identities)

   (0) ...   ...  ...  .   ...   ...    (initial state)
       C..   ...  ...  .   ...   ...    WRITE requests   C
       C1C   C..  ...  .   ...   ...    Sieve s3 expands, and propagates C
       C1C   C..  ...  C   C..   ...    COPY  propagates C
       C1C   C..  ...  C   CC.   ...    FBY   propagates C
       C1C   C..  ...  C   CD.   ...    constant delivers D
       C1C   C..  ...  D   D..   ...    FBY   passes D back
       C1D   D1.  .1.  A   A..   ...    COPY  passes D back, generating A
       C1D   D1.  .1.  .   .K.   ...    FBY propagates A, constant "dies"

   (1) D1A   A1.  .1.  .   .K.   ...    Sieve passes D back, generating A
       D1.   ...  .1.  .   .K.   ...    COPY  resolves A
       A1.   ...  .1.  .   .K.   ...    WRITE accepts D, and requests A
       .1.   ...  .1.  .   .K.   ...    Sieve resolves A, cycle finished

   (2) C1.   ...  .1.  .   .K.   ...    WRITE requests   C
       C1C   C..  .1.  .   .K.   ...    Sieve propagates C
   (3) C1C   C..  .1.  C   CK.   ...    COPY  propagates C
       C1C   C..  .1.  C   CKC   C..    FBY   propagates C
       C1C   C..  C1.  C   CKC   CCC    PLUS  propagates C
       C1C   C..  D1.  C   CKC   CDD    constant and COPY deliver D
       C1C   C..  .1.  C   CKD   D..    PLUS  passes D back
       C1C   C..  .1.  D   DK.   ...    FBY   passes D back
       C1D   D1.  .2.  A   AK.   ...    COPY  passes D back, generating A
       C1D   D1.  .2.  .   .KA   A..    FBY   propagates A
       C1D   D1.  A2.  .   .K.   .AA    PLUS  propagates A
       C1D   D1.  .1.  .   .K.   ...    COPY and constant resolve A
       if Sieve m1 finds the daton to be prime, jump to (1), else:
       C1A   A1.  .1.  .   .K.   ...    Sieve generates A (to get next daton)
       C1.   ...  .1.  .   .K.   ...    COPY  resolves A
       C1C   C..  .1.  .   .K.   ...    Sieve generates renewed C
       jump to (3)
```

Log of $\boxed{\text{Sieve}}$

Obviously, there is more to the state of the $\boxed{\text{Sieve}}$ than described by "*a*?*b*". As with the main program, we can log the state transitions during execution of the $\boxed{\text{Sieve}}$ (*a* and *b* are retained as its outport/inport labels):

```
FBY         COPY          Sieve WVR  NE   MOD FIRST (acts)
s0,   s1,s5,s8,s10,s2,    s3,   s4,  s6,  s7,  s9   (actors)
ajk  j?? n?? r?? t?? b   k?m   mnp  pqu  qrs  st    (identities)
  (4)
...  ... ... ... ... .   ...   ...  ...  ...  ..   (initial state)
C..  ... ... ... ... .   ...   ...  ...  ...  ..   demander issues 1st C
CC.  C.. ... ... ... .   ...   ...  ...  ...  ..   FBY  propagates C
CC.  C.. ... ... ... C   ...   ...  ...  ...  ..   COPY propagates C
CC.  C.. ... ... ... D   ...   ...  ...  ...  ..   supplier delivers D
CD.  D1. .1. .1. .1. A   ...   ...  ...  ...  ..   COPY accepts D, issues A
CD.  D1. .1. .1. .1. .   ...   ...  ...  ...  ..   supplier resolves A
D..  .1. .1. .1. .1. .   ...   ...  ...  ...  ..   FBY  passes back D
A..  .1. .1. .1. .1. .   ...   ...  ...  ...  ..   demander accepts D, issues A
.K.  K1. .1. .1. .1. .   ...   ...  ...  ...  ..   FBY  transforms A into K
.K.  K.1 .1. .1. .1. .   ...   ...  ...  ...  ..   COPY resolves K
  (5)
CK.  K.1 .1. .1. .1. .   ...   ...  ...  ...  ..   demander issues 2nd C
CKC  K.1 .1. .1. .1. .   C..   ...  ...  ...  ..   FBY   propagates C
CKC  K.1 .1. .1. .1. .   C1C   C..  ...  ...  ..   Sieve s3 propagates C
CKC  K.1 .1. .1. .1. .   C1C   C.C  C..  ...  ..   WVR   propagates C
CKC  K.1 .1. .1. .1. .   C1C   C.C  CCC  C..  ..   COPY delivers, NE prop C
CKC  K.1 .1. C1. .1. .   C1C   C.C  CCD  CCC  C.   const deliv;  MOD prop C
CKC  K.1 .1. D1. C1. .   C1C   C.C  CC.  CDC  CC   COPY deliv; FIRST prop C
CKC  K.1 .1. .1. D1. .   C1C   C.C  CC.  C.C  CD   COPY  delivers D
CKC  K.1 .1. .1. K1. .   C1C   C.C  CC.  C.D  DK   FIRST passes back D
CKC  K.1 .1. .1. K.1 .   C1C   C.C  CD.  D..  .K   COPY res K; MOD pass bk D
CKC  K.1 .1. .1  K.1 .   C1C   C.D  D..  ...  .K   NE    passes back D (FALSE)
  (6)
CKC  K.1 A1. .1. K.1 .   C1C   CAA  A..  ...  .K   WVR   asks for next daton
CKC  K.1 ... .1. K.1 .   C1C   C..  .AA  A..  .K   COPY  res A; WVR prop A
CKC  K.1 ... A1. K.1 .   C1C   C..  ...  .AA  AK   const res A; NE  prop A
CKC  K.1 ... ... K.1 .   C1C   C..  ...  ...  .K   COPY+FIRST res A
  (7)
CKC  K.1 ... ... K.1 .   C1C   C.C  C..  ...  .K   WVR  prop C
CKC  K.1 ... ... K.1 .   C1C   C.C  CCC  C..  .K   NE   prop C
CKC  K.1 ... C.. K.1 .   C1C   C.C  CCD  CCC  CK   MOD  prop C; const deliv D
CKC  K.1 ... C.. K.1 C   C1C   C.C  CC.  CCD  DK   COPY prop C; FIRST deliv D
CKC  K.1 ... C.. K.1 D   C1C   C.C  CC.  CC.  .K   supplier deliv D
CKC  K.1 .1. D1. K.1 A   C1C   C.C  CC.  CD.  .K   COPY  pass bk D
CKC  K.1 .1. .1. K.1 .   C1C   C.C  CD.  D..  .K   MOD pass bk D; supplier res A
CKC  K.1 .1. .1. K.1 .   C1C   C.D  D..  ...  .K   NE   passes back D
  if NE delivers the daton value FALSE jump to (6), else jump to (8):
              — continued —
```

— continued —

```
FBY        COPY           Sieve WVR  NE   MOD FIRST (acts)
s0,   s1,s5,s8,s10,s2,   s3,  s4,  s6,  s7,  s9   (actors)
ajk  j?? n?? r?? t?? b   k?m  mnp  pqu  qrs  st   (identities)
  (8)
CKC  K.1 C1.  .1.  K.1 .  C1C  CC.   ...   ...  .K  WVR requests C from left op
CKC  K.1 D1.  .1.  K.1 .  C1C  CD.   ...   ...  .K  COPY delivers D
CKC  K.1 .1.  .1.  K.1 .  C1D  D..   ...   ...  .K  WVR  passes back D
  if Sieve s3 finds the daton to be prime jump to (9), else:
CKC  K.1 .1.  .1.  K.1 .  C1A  A..   ...   ...  .K  Sieve asks for next daton
CKC  K.1 A1.  .1.  K.1 .  C1.  .AA   A..   ...  .K  WVR prop A
CKC  K.1 ...  .1.  K.1 .  C1.  ...   .AA   A..  .K  NE  prop A; COPY  res A
CKC  K.1 ...  A1.  K.1 .  C1.  ...   ...   .AA  AK  MOD prop A; const res A
CKC  K.1 ...  ...  K.1 .  C1C  C..   ...   ...  .K  COPY+FIRST res A; Sieve prop C
  jump to (7)
  (9)
CKD  K.1 .1.  .1.  K.1 .  D1A  A..   ...   ...  .K  Sieve s3 pass bk D, gen A
CKD  K.1 A1.  .1.  K.1 .  D1.  .AA   A..   ...  .K  WVR prop A
CKD  K.1 ...  .1.  K.1 .  D1.  ...   .AA   A..  .K  NE  prop A; COPY  res A
CKD  K.1 ...  A1.  K.1 .  D1.  ...   ...   .AA  AK  MOD prop A, const res A
CKD  K.1 ...  ...  K.1 .  D1.  ...   ...   ...  .K  COPY+FIRST res A
DK.  K.1 ...  ...  K.1 .  .1.  ...   ...   ...  .K  FBY   passes back D
AK.  K.1 ...  ...  K.1 .  .1.  ...   ...   ...  .K  demander accepts D, gen A
.KA  K.1 ...  ...  K.1 .  A1.  ...   ...   ...  .K  FBY   propagates A
.K.  K.1 ...  ...  K.1 .  .1.  ...   ...   ...  .K  Sieve s3 resolves A
  (10)
CK.  K.1 ...  ...  K.1 .  .1.  ...   ...   ...  .K  demander issues C
CKC  K.1 ...  ...  K.1 .  C1.  ...   ...   ...  .K  FBY   propagates C
CKC  K.1 ...  ...  K.1 .  C1C  C..   ...   ...  .K  Sieve prop C
  jump to (7)
```

Discussion

These substantial logs demonstrate a number of things quite clearly:

We see how the requests (**C** and **A**) ripple upstream. They appear as lines ***falling from left to right***, since the outports and high ranking actors are on the left and the inports and low ranking actors are on the right. Similarly one can see how replies (**D**) flow downstream; they form lines ***falling from right to left***.

The log gives numerous illustrations for the behaviour of COPY node actors. Whenever a daton value (**D**) is accepted by a COPY node actor, COPY queues it at all its outports, and sends an ADVANCE request (**A**) upstream. Once a daton has been queued at COPY it can be obtained from there, any number of times. The main program log shows furthermore how COPY can satisfy one outport *while* another COPY outport is hung waiting for a new daton. (Our main program contains a cycle, cycles are always "tapped" by a COPY, and every cycle–tapping COPY must handle such *interlaced* requests.)

The log quotes the *length of every queue* at every moment. In our LUX implementation, we used a shared queue with reference counts. This shared queue is at every moment equal to the *longest* of all the individual queues.

We see how FBY, upon receiving its first proper **A** request, abandons (**K**) its left operand and switches over to its right operand. FIRST goes even further: it abandons its operand upon *arrival* of the first daton.

The work of the WRITE actor m0 consists obviously in repeatedly

(1) issuing a **C** request to the Sieve m1,

(2) awaiting daton delivery,

(3) printing the daton value,

(4) issuing an **A** request.

Hypothetically, if WRITE chose to skip (4) it would get exactly the previous daton again. WRITE can get at the next daton in the history (the next prime number) only after sending an ADVANCE request (**A**) to the Sieve m1. Upon this **A** request, a clean–up is carried out. The log shows how **A** is propagated upstream, but shows also that COPY does not propagate the **A** further. COPY had *anticipated* this **A** already upon receipt of the daton. The Sieve (having a COPY at its inport) behaves likewise.

Whenever a COPY node actor issues a **C** request, it does this in response to the arrival of a **C** request at an outport. (Not every **C** request at an outport is propagated by COPY.) This outport is then called the **driving outport**. In the Sieve main program, for example, the left COPY outport is always the driving one. However, the role of driving outport need not always fall to the same outport. In the Sieve UDF, for example, this role is first taken by $\boxed{\text{COPY}}$ outport s1 and later by s8. While an outport is driving, its queue can only be empty or of length one.

Focussing more specifically on the $\boxed{\text{Sieve}}$ UDF, a few points deserve mention:

One can formulate an obvious theorem: "the computation of any daton must be carried out in a ***finite*** number of steps; otherwise the computation is in a ***livelock***". This implies that one daton must require only finitely many UDF expansions. The $\boxed{\text{Sieve}}$ UDF satisfies this clearly: the newly expanded $\boxed{\text{Sieve}}$ computes its first daton without expanding any further $\boxed{\text{Sieve}}$, and after that, exactly one new $\boxed{\text{Sieve}}$ is created whenever a new prime (= result daton) has been found.

The log reveals also that, before the $\boxed{\text{Sieve}}$ can deliver a daton, it must consume at least ***one*** daton from its supplier. One can trace daton deliveries simply by scanning down a log column until one comes to a point where it changes from **C** to **D** and then to **A**. In the case of the $\boxed{\text{Sieve}}$ UDF, the outport is labelled ***a*** and its inport ***b***; these are therefore the log columns of interest.

The example demonstrates hardly any "tricky" situations: there are only few bare $\boxed{\text{ADVANCE}}$ requests, and no $\boxed{\text{COMPUTE}}$ request is nullified. The example appears even to be ***deterministic***, but this is not the case. The log shows the states as if every transition was made at the ***earliest possible*** moment. In reality, however, each actor is free to ***delay*** its action for any period. The log would be of enormous size if all the possible alternatives had been included in it.

CHAPTER VI: Ways of Improving Efficiency

6.0 Introduction

Any Lucid program can be built up node by node, starting at the $\boxed{\text{WRITE}}$ node. During this construction, each intermediary structure can be examined for specific properties. This chapter will show that, under various conditions, structures can be replaced by simpler ones. Simplicity means often smaller overheads, less administration (though on cost of generality), and less administration means in most cases faster execution.

We shall look in this chapter at various code improvement techniques:

- Queuing analysis (Cycle Sum Test),
- Node condensing (act expansion),
- Enriching the protocol,
- Tailoring $\boxed{\text{COPY}}$ acts,
- Tagged Data Flow, and
- "Box of tricks" for the compilation.

Beginning with this chapter, matters will be treated less formally: the general method will be sketched while the detail will be left to later research.

6.1 Queuing Analysis

It is characteristic of our (demand driven) implementation that all the daton buffering is done by $\boxed{\text{COPY}}$ node actors. However, the $\boxed{\text{COPY}}$ act (↑ 4.6.7 ff) makes only too clear how much administration is entailed even in very simple operations. There are enough situations where a much *simpler* $\boxed{\text{COPY}}$ node actor would suffice:

- there may be an ***upper bound*** for the queue length,
- the offset between the outport indices may be ***invariant*** (and the driving outport may be always the same),
- the buffering may be unnecessary altogether.

Sections 6.4 and 6.5 outline $\boxed{\text{COPY}}$ acts which can exploit such special conditions. The log of the $\boxed{\text{Sieve}}$ program (↑ 5.7) illustrated the growing and shrinking of the queues quite vividly. One could now extend the Lucid compiler by a simulation phase which generates logs, and which detects the queuing behaviour in this way. Such a device would provide the optimiser with all imaginable facts, but it would be a very complex program (also very slow in execution). Anyway, there is a much simpler method which provides almost the same answers. The method is the *index offset* method (derived from Wadge's Cycle Sum Test [Wad79]) which will be described now.

Index Offset and Offset Matrix

Focussing on a particular port of an actor, it is possible, at every moment of program execution, to state the index of the daton currently due to traverse that port (say, upon a $\boxed{\text{COMPUTE}}$ request). Initially, every index is zero. As program execution progresses, the index increases by 1 with every $\boxed{\text{ADVANCE}}$ request traversing the port. Input histories are gradually consumed and output histories are produced, and we see the indices at inports and outports grow, more or less synchronously. For example, in a *pointwise* node actor (e.g. $\boxed{\text{PLUS}}$ or $\boxed{\text{IF}}$) the index is the same on all ports (if we ignore intermediary states). However, at $\boxed{\text{FBY}}$ actors the index at the right inport lags by 1 behind the outport:

FBY ports	Daton index					→ time		
outport	0	1	2	3	4	5	6	...
inport	0	∞	∞	∞	∞	∞	∞	...
inport	0	0	1	2	3	4	5	...

The left FBY inport "dies" at the first ADVANCE request, its index jumps to finalindex (we write "∞"). With NEXT it is the other way round, its inport is one ahead of the outport:

NEXT ports		Daton index						→ time	
outport	g	0	0	1	2	3	4	5	...
inport	p	0	1	2	3	4	5	6	...

These tables for FBY and NEXT suggest how to characterise the behaviour of the port indices. We use a *matrix* with as many rows as the node has outports, and as many columns as the node has inports. Each component of this matrix is called an **index offset**, and it is defined:

$$\text{index offset} = \underset{\text{all indices}}{\text{MINIMUM}} \left(\text{outport index} - \text{inport index}\right)$$

Included in this minimum are only the situations where datons actually traverse both the inport and the outport (the 0–0 of NEXT is thus omitted). At least for the fundamental operators, except COPY, such an **offset matrix** is easy to write. Each component of the matrix is either an integer or "−∞". A component of value "−∞" marks those inports from which an outport is totally independent.

Most nodes have only one outport, and the matrix has only one single row there. The matrices for constant, READ and WRITE nodes have no components. The matrices for *pointwise* nodes have only components of value 0. The matrices for FBY and NEXT are:

$$\text{FBY:} \quad \begin{bmatrix} 0 & -1 \end{bmatrix} \qquad \text{NEXT:} \quad \begin{bmatrix} +1 \end{bmatrix}$$

There is no corresponding easy rule for the indices at COPY. Its inport index is usually the maximum of its outport indices, but this rule does not apply strictly. (The strict rule goes as follows: we mark a COPY outport as *bare* whenever it gets a bare ADVANCE request or an ADVANCE, finalindex. Initially, and after a proper ADVANCE, this mark is cleared. The inport index is the maximum of the unmarked outport indices, *if any*; otherwise it is the maximum of all outport indices.) However, there are other ways of finding out the $\boxed{\text{COPY}}$ inport indices. Every $\boxed{\text{COPY}}$ node actor is connected to ***other*** node actors, and from the indices of these other actors one can derive the indices at the $\boxed{\text{COPY}}$ itself. We can ***provisionally*** assume that the outports of $\boxed{\text{COPY}}$ never go bare, and apply a strict maximum rule for its inport index; consequently, all its index offsets will come out as non-negative. At every moment, one component of the $\boxed{\text{COPY}}$ offset matrix is usually zero (viz. the one of the driving outport).

Intuitive Meaning of Index Offsets

The meaning of index offsets has not been made clear yet. There are, however, many analogies at hand for illustrating it.

Data Flow computations are organised a bit like the work on a production line (in a somewhat old-fashioned factory): the components are accepted from preceding production processes, operations are applied to these components, and the resulting item is passed on further down the line.

Our old analogy of the small restaurant "*Chez Lucien*" can help; after all, restaurants ***produce*** meals. Pots and pans are needed for the cooking, but our cook has only few of them. The cooking of a meal can begin only after the pots and pans from an ***earlier*** meal have been washed up. For the ***first*** meal(s), however, the pots and pans are taken straight from the shelves. One can say, the pots and pans are "***injected*** into the production cycle" to start things up.

In Data Flow, as well as in these analogies, work can go on only if all the ***prerequisites*** are available. There can never be ***too many*** prerequisites, i.e. it does not interfere if the prerequisites for later work queue up, as this creates merely some "**slack**" in the schedule. If there are many kinds of prerequisites, the supply with the ***least*** slack determines whether production goes on or not. Slack matters most in ***cyclical*** production processes, i.e. where the supply for a production process is somehow conditional on earlier output of the same process (like re–using pots and pans). Indeed, work can go on only if there is a modicum of slack: each production step takes the prerequisites (***removes*** one unit of slack), manipulates them, and outputs them (***restores*** one unit of slack).

Every $\boxed{\text{FBY}}$ node ***creates*** slack, it injects one daton into the stream flowing from its ***right*** inport to its outport (no extra slack on its left inport). Every $\boxed{\text{NEXT}}$ node ***removes*** slack, it eliminates one daton from the stream. The offset matrices express the provision or removal of slack on the stream flowing from an inport to an outport. ***Negative*** index offsets indicate the ***provision*** of slack, and vice versa. (This definition is now standard, for mathematical reasons [AsW83]. Early publications [Wad79] used the opposite definition.)

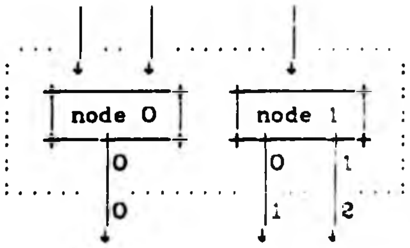
At some moments, however, there may be more slack than busy nodes to use it up. This means the slack has to be accommodated somewhere else, namely in a $\boxed{\text{COPY}}$ node. Its buffer queue takes up the remaining slack, and the queue length is therefore less or equal the (inverse of the) index offset. $\boxed{\text{COPY}}$ nodes provide no cure if there is insufficient slack.

Wadge's [Wad79] **Cycle Sum Test** states (in essence) that every cycle is certainly free from deadlock if it has slack of at least one daton. A cycle with fewer $\boxed{\text{NEXT}}$s than $\boxed{\text{FBY}}$s passes the Cycle Sum Test. The Cycle Sum Test gives merely a ***worst case*** analysis (the rule is sufficient, but not necessary), whereas logs give the whole answer (only: to produce the complete log may take very long, possibly forever). Further below we will see how particular constellations of nodes permit a relaxation of the rule. – The Cycle Sum Test determines the ***minimum*** queue length, and requires it to be at least one. However, in order to optimise $\boxed{\text{COPY}}$ (↑ 6.4), one needs to know the ***maximum*** queue length. This can, still, be found by index offset methods (↑ "***iteration***", below).

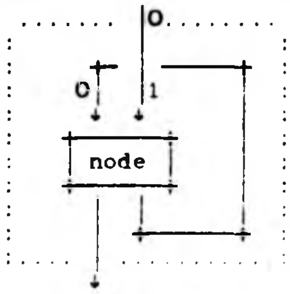
Any serious computation involves a large number of nodes, and one would wish to know how much slack there is in such a large composite. This can be achieved by matrix operations, to be described now.

Net Construction

An arbitrary Lucid graph (a net or a subnet) may be constructed using merely two tools: ***juxtaposition*** and ***iteration*** (see also [Fau82], pages 140 ff). At each construction step, we can determine the offset matrix of the object built up so far.



Juxtaposition



Iteration

Juxtaposition is the operation which takes two arbitrary nodes, and places them side–by–side. **Iteration** is the operation which takes an arbitrary node, and connects a particular inport to a particular outport. Both operations re–index the inports and outports in the obvious way.

The number of inports of the juxtapositioned super–node is the sum of inports of its inner nodes, and the same is valid for the outports. The offset matrix of the super–node is obtained by placing the offset matrices along the main diagonal, and by padding the remainder with "$-\infty$". For the example above (let a, b, c, d be offsets):

node 0 : $\begin{array}{|cc} a & b \end{array}$ node 1 : $\begin{array}{|c} c \\ d \end{array}$ juxtaposition: $\begin{array}{|ccc} a & b & -\infty \\ -\infty & -\infty & c \\ -\infty & -\infty & d \end{array}$

The number of inports of the *iterated* super–node is one less than that of its inner node, and the same is valid for the outports. The offset matrix of the super–node is obtained by forming the *maximum* of the index offsets along the paths which connect the new outport to the new inport. The matrix component for the inport and outport to be connected must be < 0 (otherwise the net *may* deadlock). For the example above:

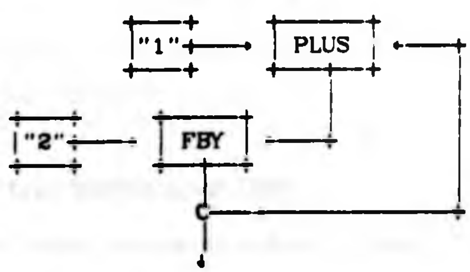
inner node : $\begin{array}{|cc} a & b \\ c & d \end{array}$ iteration: $\begin{array}{|l} \max\{a+d,\ b\} \end{array}$

$$(c \text{ must be } < 0 \qquad s = |a+d - \max| + |b - \max| = |a+d - b|\ )$$

The condition "$c < 0$" is due to the Cycle Sum Test; "s" expresses how much total slack there is on the node–internal paths.

Example (Sieve main program): offset matrix

We are now able to apply this method, for example, to a subnet which occurred in the main program for the Sieve:



We want to work out the queue lengths in the COPY node at the bottom. The offset matrix of the PLUS node consists only of zeros, and the matrix of the PLUS node compounded with the "1" is just a zero. The matrix of the FBY compounded with the

"2" is just a −1, and the entity consisting of "2", FBY, "1" and PLUS gives therefore an offset matrix of just a −1. Yet, we know nothing about the matrix of our COPY.

2 FBY 1+...	COPY :	Juxtaposition:	
−1	*c*	−1	−∞
	d	−∞	*c*
		−∞	*d*

We connect (= iterate) the outport of the compound to the COPY inport, and get:

$$\begin{array}{|l} \hline c-1 \\ d-1 \end{array}$$

and connect the outport of the whole thing to the right COPY outport. This leads to a matrix without any columns (since there are no inports), but there is also the condition "$d-1 < 0$". This condition states that the COPY node must provide buffer space for (up to) ***one*** daton on its right outport. The left COPY outport is the only subnet outport (it must be therefore the ***driving*** outport for the whole subnet), and its queue length is therefore ***zero***.

But this is not all. The queue lengths were calculated under the assumption that we are not in an ***intermediary state***. In other words, it describes the state where the daton delivery has been acknowledged by an ADVANCE request. During the evaluation of a daton the COPY queues can swell to a length which is one greater than calculated above. In our example, above, the left COPY outport must thus provide a buffer for one daton, and the right COPY outport must provide space for two, and this is indeed in harmony with the log (↑ 5.7).

Offset Matrices of UDFs

The offset matrix of a ***non–recursive*** UDF can be determined just by applying juxtaposition and iteration, as described.

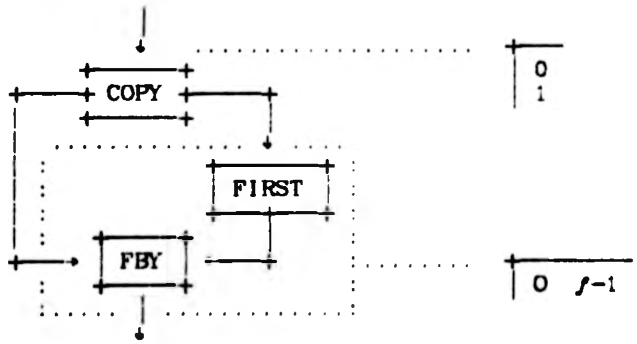
Matters are more difficult for a ***recursive*** UDF, since its offset matrix is defined ***in terms of itself***. It is nevertheless quite straight forward to compute. We apply our usual construction process (juxtaposition and iteration), though using a matrix of ***unknowns*** for the recursive UDF. Once the construction is done, we can equate the resulting offset matrix with the matrix used at the outset, and we are left with a system of ***linear*** equations or inequalities. The solution of this system is the offset matrix.

Example (FIRST)

This process can be illustrated using a recursive definition of FIRST:

```
FIRST a  =  a FBY (FIRST a) ;
```

The corresponding Lucid graph is (ignore the offset matrices for a moment):



The offset matrix of FIRST has only one component, namely f, the subnet of FBY and FIRST (framed in dots in the drawing) has the offset matrix $<0, f-1>$. We iterate this subnet with the COPY node, and have to form the maximum of the offset matrix above (the offset matrix of our COPY node is yet unknown). The end result is a FIRST node again, and we equate therefore that maximum with f itself:

$$f \;=\; \max \{\, 0,\; f-1 \,\}$$

The solution is $f = 0$, which means the sole component of the offset matrix for the UDF $\boxed{\text{FIRST}}$ is zero. We conclude that our COPY has the offset matrix $<0,\ 1>^T$, as shown. It has therefore buffer space for one extra daton on the right outport and none on the left (actually one more each, to allow for intermediary states).

The same process can be applied to the recursive UDF $\boxed{\text{WVR}}$, and we would find that both components of its offset matrix are ∞. The sole component of the offset matrix of the recursive UDF $\boxed{\text{Sieve}}$ is also ∞. This means that, in either UDF, the inport indices may be arbitrarily far ahead of the outport index, and its actor might therefore have ***unbounded*** buffering needs. We know from the log, however, that ***each invocation*** of the $\boxed{\text{Sieve}}$ retains one daton; there is one invocation per prime. If we made a log of $\boxed{\text{WVR}}$, we would find that the top invocation retains one daton whereas all earlier invocations have no memory. – By the way, the offset matrix of $\boxed{\text{UPON}}$ is $<0,\ -1>$ and, again, the top invocation retains one daton whereas all earlier invocations have no memory.

6.2 Act Expansion, and Node Condensing

The translation process described in chapter IV leaves us with a large number of actors (viz. one actor per node). This is exactly what is needed for a computer of the latest design, many cheap processors closely coupled together. In traditional computer systems, however, concurrency must be restricted to those cases where it is essential; some uses of the ***concurrent*** *OR* are such essential cases. (Real time requirements provide an unending supply of further examples, but that topic goes beyond this thesis.) We turn our attention in this section to an optimisation for a setting where concurrency must be minimal. The optimisation technique of this section is not applicable where the superior is a concurrent operator or where the inferior has more than one reference (=$\boxed{\text{COPY}}$). – We prepare the program for this optimisation by replacing concurrent operators by their non-concurrent counterparts wherever possible (i.e. wherever that is not against the idea of the program).

As far as this section is concerned, three parts of every node act (↑ end of 4.1) are particularly relevant: X–part, Y–part and exception part. The ***X–part*** is executed only once at the beginning (the actor initialisation is part of this), the ***Y–part*** is executed in every loop pass, and the ***exception part*** is executed in the event of an exception.

When a node actor gets a COMPUTE or ADVANCE request, it resolves it within exactly one loop pass, i.e. by executing the Y–part or the exception part. An undirected RECEIVE is the first instruction in the Y–part, and this is exactly where the actor accepts ***all*** COMPUTE requests. Some acts ("***finite state machines***", such as FBY or NEXT) do not fit into this layout right away, but they can be brought into the universal shape with the help of CASE statements.

On the side of the superior, a mere ***two*** pieces of code produce the requests. The daton value is acquired from the operand actor by calling GetDaton (which issues also NULLIFY, if required), and an EXCEPTION ADVANCE does the rest. (Concurrent acts use generally means other than GetDaton, which is why our optimisation mechanism cannot be applied there.)

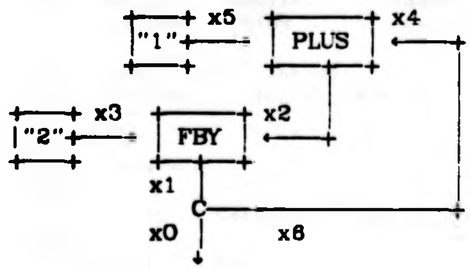
The optimisation is easily carried out: just append the inferior X–part to the X–part of the superior, substitute the call to GetDaton by the inferior Y–part, and substitute the EXCEPTION ADVANCE by the inferior exception part. – Clearly, this transformation is an ***expansion*** (* 4.3.2). It has no effect on the computations, but it reduces the number of actors and the amount of message passing. The expansion is, of course, hardly possible if the inferior has more than one reference (sole example: COPY inport actor). It goes without say that no law forbids the expansion of expanded code; expansion may indeed be re–applied up to any finite depth.

The code of a UDF act *itself* (↑ 4.3.3.1) can be put in place of the UDF reference (i.e. expanded), but the instructions inside the UDF act ($\boxed{\text{CREATE}}$ etc.) must clearly not be expanded (*lazy expansion* must be maintained, ↑ 4.3.3.1). UDF actors are, like all node actors, usually accessed from more than one point (at least from $\boxed{\text{GetDaton}}$ and from $\boxed{\text{EXCEPTION ADVANCE ...}}$). The UDF expansion must be programmed with care so that one UDF subnet is not created more than once. – If a UDF is unlikely ever to get a $\boxed{\text{COMPUTE}}$ request, it can be advantageous to leave it *unexpanded*, even if the UDF is non–recursive. It will use hardly any space until it gets its first $\boxed{\text{COMPUTE}}$ request.

Act expansion has its drawbacks. Without it, the program would use mainly the standard acts (↑ 4.5), and only the UDF acts would have to be defined and compiled individually. The *shared* use of (standard) acts keeps the memory requirements low. As soon as one standard acts is expanded and integrated into another standard act, we end up with one act more, which has its price. One has to weigh the number of actors against the number of acts. Generally, act expansion is indicated if it greatly reduces the number of actors. It is of real benefit only if applied to a depth *much* greater than one.

Example (Sieve main program): act expansion

The subnet from the Sieve main program is well suited to demonstrate act expansion. Here is the subnet once more:



We start our analysis at the subnet *outport* (as did the translation algorithm of section 4.3.4), namely the outport actor x0 of COPY. Its inferior x1 cannot be expanded (at least not straight forward) since it has two superiors; we leave x0 unchanged for the time being and move on to its inferior x1. Its inferior x2, the FBY actor, can be expanded. The relevant portions in actor x1 are:

```
(* Declarations:                            Version 0 *)
   LABEL 1 ;
   VAR   request : MSGTYPE ;     index  : INTEGER ;
         x2      : ACTOR   ,     result : ANYTYPE ;

(* X-part and initialisation:                         *)
   ( , , x2) := RECEIVE FROM (Creator) ;

(* Y-part:                                            *)  :1
   result := GetDaton (x2, index) ;

(* Exception part:                                    *)
1: (request, index) := Reveal ;
   IF   request = ADVANCE
   THEN EXCEPTION (request, index) TO (x2) ;
   RESET ;
```

We expand the FBY actor x2 and obtain a rather clumsy piece of code (this is essentially Act_Fby_1):

```
(* Declarations:                                    Version 1 *)
   LABEL 1 ;
   VAR   request : MSGTYPE ;     index  : INTEGER ;
         x3, x4  : ACTOR   ;     result : ANYTYPE ;

(* X-part and initialisation:                                 *)
   ( , , x3, x4) := RECEIVE FROM (Creator) ;

(* Y-part:                                                    *)
   IF   index = 0
   THEN                                                         :1
        result := GetDaton (index,   x3)
   ELSE                                                         :1
        result := GetDaton (index-1, x4) ;

(* Exception part:                                            *)
1: (request, index) := Reveal ;
   IF   request = ADVANCE
   THEN BEGIN
          IF        index = 1
          THEN      EXCEPTION (request, finalindex) TO (x3)
          ELSE IF   index = finalindex
               THEN EXCEPTION (request, index     ) TO (x3, x4)
               ELSE EXCEPTION (request, index - 1 ) TO     (x4) ;
        END ;
   RESET ;
```

The left operand of the FBY is a constant, and our code becomes a good deal simpler by expanding Act_Constant:

```
   LABEL 1 ;                  (* Declarations.   Version 2 *)
   VAR   request : MSGTYPE ;     index  : INTEGER ;
         x4      : ACTOR   ;     result : ANYTYPE ;

(* X-part and initialisation:                            *)
   ( , , x4) := RECEIVE FROM (Creator) ;

   IF   index = 0             (* Y-part.                 *)
   THEN result := 2
   ELSE                                                    :1
        result := GetDaton (index-1, x4) ;

1: (request, index) := Reveal ;  (* Exception part. *)
   IF   (request = ADVANCE)    AND    (1 <> index)
   THEN BEGIN
          IF   index = finalindex
          THEN EXCEPTION (request, index  ) TO (x4)
          ELSE EXCEPTION (request, index-1) TO (x4) ;
        END ;
   RESET ;
```

It is also easy to expand the right operand of FBY, the PLUS node. The "index fiddle" of FBY carries through to the operands of PLUS. We expand the left operand x5 as we did before with x3, and we get:

```
   LABEL 1 ;                  (* Decl.       Version 3 *)
   VAR   request : MSGTYPE ;     index  : INTEGER ;
         x6      : ACTOR   ;     result : ANYTYPE ;

(* X-part and initialisation.                            *)
   ( , , x6) := RECEIVE FROM (Creator) ;

   IF   index = 0             (* Y-part.                 *)
   THEN result := 2
   ELSE                                                    :1
        result := 1 + GetDaton (index-1, x6) ;

1: (request, index) := Reveal ;  (* Exception part. *)
   IF   (request = ADVANCE)    AND    (1 <> index)
   THEN BEGIN
          IF   index = finalindex
          THEN EXCEPTION (request, index  ) TO (x6)
          ELSE EXCEPTION (request, index-1) TO (x6) ;
        END ;
   RESET ;
```

The next step would be to expand the $\boxed{\text{COPY}}$ outport actor $\boxed{x6}$, but we hesitate here. Firstly, the code of $\boxed{x6}$ is too massive to benefit when expanded; secondly, we cannot apply expansion any further because of $\boxed{x1}$ having two superiors. Thirdly, if we did nevertheless expand $\boxed{x6}$ and integrate it in $\boxed{x1}$, the $\boxed{\text{COPY}}$ inport would end up trying to request from itself. However, LUX actors cannot exchange messages with themselves. On the other hand, we found in section 5.7 that $\boxed{x0}$ is always the *driving* outport, and that the queue length on the right inport is one. This is why $\boxed{x6}$ need never exchange messages with $\boxed{x1}$, and expanding $\boxed{x6}$ would therefore not cause any problem.

To bring the example to a conclusion, let me anticipate a tailor—made $\boxed{\text{COPY}}$ act with just the right properties (with a "cyclic" buffer of size one) which will be presented in section 6.4. Special attention has been paid to making sure the act can handle *bare* $\boxed{\text{ADVANCE}}$ requests properly. We expand the constituents of that $\boxed{\text{COPY}}$ act, and get one node act for the *entire subnet*:

```
ACT Act_x0 ;                                                   Version 4

    LABEL 1 ;           (* Declarations                        *)
    VAR   superior    : ACTOR    ;     request : MSGTYPE ;
          now, index  : INTEGER  ;     result  : ANYTYPE ;

    BEGIN
    now :=  1 ;         (* X-part                              *)

    REPEAT
      WHILE TRUE DO
      BEGIN                                                       :1
        (superior, request, index) := RECEIVE () ;

        (* Y-part:                                             *)
        WHILE  now < index
        DO BEGIN                                                  :1
             now := now + 1 ;
             IF   now = 0
             THEN result := 2
             ELSE result := result + 1 ;
           END ;
                                                                  :1
        SEND (DATON, result) TO (superior) ;
      END ;             (* End of inner eternal loop.           *)

1:    RESET ;           (* Exception part.                      *)
    UNTIL FALSE ;       (* End of outer eternal loop.           *)
    END ;
```

The cell now retains the last index for which the result has been computed, and the evaluation does some "catching up" (WHILE now < index) when required. This elaborate mechanism has been inherited from the tailor–made COPY act; it needed this mechanism for handling bare ADVANCE exceptions correctly.

An optimising compiler could go a step further. It has been mentioned that some acts must be conditioned to be suitable for expansion (* beginning of 6.2, "finite state machine"). However, once expansion has been carried out to exhaustion, the ***reverse*** conditioning can be attempted. If the Y–part handles the evaluation of its ***initial*** daton differently from the rest, it may help to **unwind** this initial loop pass (make a copy of the loop body, specialise it for one index value), and to place it before the loop (i.e. append it to the X–part). This process may be applied repeatedly.

One loop pass (viz. setting of the starting value) can be unwound in our example. Very little computation is actually carried out from one index value to the next, and the computation could therefore be done in the exception part. Some reorganisation of the program results in:

```
ACT Act_x0 ;                                              Version 5

    LABEL 1 ;            (* Declarations                       *)
    VAR   superior : ACTOR   ;   request : MSGTYPE ;
          index    : INTEGER ;   result  : ANYTYPE ;

    BEGIN
    result := 2 ;        (* X-part                             *)

    REPEAT
      WHILE TRUE DO
      BEGIN                                                       :1
        (superior, request, index) := RECEIVE () ;

        (* Y-part is empty                                     *)
                                                                  :1
        SEND (DATON, result) TO (superior) ;
      END ;               (* End of inner eternal loop.  *)

1:    IF   Reveal  = ADVANCE       (* Exception part.  *)
      THEN result := result + 1 ;
      RESET ;
    UNTIL FALSE ,         (* End of outer eternal loop.  *)
    END ,
```

Even better, the compiler might detect and exploit that the result is a linear function of the daton index:

```
ACT Act_x0 ;                                              Version 6

    LABEL 1 ;               (* Declarations                   *)
    VAR   superior : ACTOR   ;    request : MSGTYPE ;
          index    : INTEGER ;    result  : ANYTYPE ;
    BEGIN                   (* X-part is empty                *)

    REPEAT
      WHILE TRUE DO
      BEGIN                                                      :1
        (superior, request, index) := RECEIVE () ;

        (* Y-part:                                            *)
        result := index + 2 ;
        (*      := index * increment + start                  *)
                                                                 :1
        SEND (DATON, result) TO (superior) ;
      END ;                 (* End of inner eternal loop.     *)

1:    RESET ;               (* Exception part.                *)
    UNTIL FALSE ;           (* End of outer eternal loop.     *)
    END ;
```

Actors created from this act have no memory, and the act is therefore as easy to expand as $\boxed{\text{Act\_Const\_}}$.

6.3 Enriching the Protocol

The universal protocol (↑ 4.2) has proved just right for all the explanations so far; a more refined protocol might well have blurred the relevant issues. But we shall now study some ***protocol extensions***, most of them aimed at making better use of the $\boxed{\text{COPY}}$ node actors.

All ***replies*** were so far of message type $\boxed{\text{DATON}}$. A reply of the alternative message type $\boxed{\text{CONSTANT}}$ could imply that all later replies will have the same value. $\boxed{\text{IF}}$ and $\boxed{\text{COPY}}$, even $\boxed{\text{NEXT}}$ and $\boxed{\text{FBY}}$, could take advantage of this extra information. It is unfortunately not easy to recognise ***all*** structures which deliver constants. – Occasionally, actors have to switch into the "***through***" mode, where all subsequent requests and replies are passed on unchanged. This situation could be optimised by

operand redirection, i.e. by extra information in the *reply* telling "substitute this operand from now on by actor *xyz*".

Here are some extended *requests* (all directed at actor ●):

AUGMENT?: (● = COPY outport) create a further COPY outport actor,

LENGTH?: (● = COPY outport) enquire for current queue length,

QUEUE d: (● = COPY inport) append daton *d* to the queue,

RESTART: reset ● as if it had just been created and initialised,

KILL: eradicate ● and its dedicated inferiors.

Only the first two requests get replies. The requests are listed in the order of increasing relevance, and difficulty. The list is anything but complete (further suggestions: "bulk demand" ↑ 4.6.13, and a special BARE exception in place of the *bare* ADVANCE). Let us study the extensions one by one.

AUGMENT

If one COPY node actor feeds *directly* into another COPY node actor, some wasteful buffering of datons can occur (duplication, ↑ 4.6.1). Such a configuration can occur in perfectly meaningful programs. In the Sieve program (↑ 5.7), for example, the COPY in the main program feeds straight into the COPY of the UDF. This situation can be saved by the request AUGMENT. Issued to a COPY outport ●, AUGMENT would cause ● to create a further COPY outport actor E, with E initially referring to the same daton as ● (i.e. E starts from the present state of ●). Upon the AUGMENT request, ● gives the actor name of E as reply.

LENGTH

There are numerous applications for a request LENGTH which helps to find out the ***current queue length*** of a COPY outport actor (or a READ actor). It is almost indispensable in the interface from a demand driven to a ***data driven*** evaluation.

A mixed Lucid implementation with provisions for data driven evaluation has its attractions: it can use the idle time of the processor (e.g. waiting for inputs from the user) for some "***compute ahead***", especially if this does not increase total store requirements.

QUEUE d

The QUEUE *d* request is similarly important for the interface ***from*** a data driven to the demand driven evaluation. That request, when issued to a COPY inport actor •, appends daton *d* to the buffer queue of •. (The COPY inport act of section 4.6.1.1 would need modification to accept ***requests***.) — This enhancement permits a FBY optimisation: every FBY node inserts "slack" into the daton stream and, with the help of QUEUE *d*, one would be able to "push" this daton downstream before the program start. The corresponding optimisation or NEXT requires ***no*** special means: merely a ***bare*** ADVANCE must be passed ***up***stream before the program start.

RESTART

Some recursive UDFs cause an unending need for the creation of new actors, while at the same time shedding defunct actors (see KILL request, best example: Igloo function ↑ appendix B). It is often possible (Lucid ***tail recursion***, ↑ 6.6) to immediately assign a ***new*** role to a actor •, instead of letting it die. This is achieved with the help of the RESTART request, which makes • pretend it had just been created and initialised. Actors ***propagate*** RESTART requests to their inferiors.

KILL

We have so far used ADVANCE, finalindex to tell actor • that its services are no longer needed. Upon ADVANCE, finalindex, actor • does a "last clean-up" and goes then into eternal hibernation (i.e. it does not terminate its existence; it may be followed by a RESTART request). The KILL exception exceeds the effect of

ADVANCE, finalindex in that it *does* terminate the existence of ● (and a RESTART is then impossible). Actors *propagate* KILL exceptions to their inferiors in the course of the clean-up. After the KILL, the entire subnet is eradicated. Once an actor has received a KILL no further messages must ever be sent to it, since "messages must only be sent to *existing* actors" (↑ 3.2.2, SEND). – KILL improves efficiency, obviously, since it releases resources for re-use.

When tracing upstream through the subnet, we may come to a COPY node actor which is not *entirely* dedicated to the subnet. If ● is a COPY *outport* actor, a KILL exception will certainly terminate ●, but it will terminate the pertaining COPY *inport* actor only if no other outport actors remain. This is controlled by the active voting mechanism (↑ 4.6.7) in procedure AdvanceOutport.

The (revised) exception part in the FBY act is the ultimate source of most KILL exceptions (another source is IF with computed constants, ↑ 6.6). The revised code would look roughly like this:

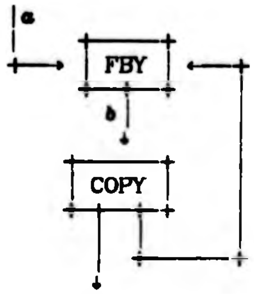
```
1: (request, index) := Reveal ;
   CASE request OF
     ADVANCE: IF   index = 1
              THEN EXCEPTION (KILL, finalindex) TO (p0)
              ELSE EXCEPTION (ADVANCE, index-1) TO (p1) ;

     KILL:    IF   index = 1
              THEN EXCEPTION (KILL, finalindex) TO (p0, p1)
              ELSE EXCEPTION (KILL,     index-1) TO     (p1) ;
     NULLIFY:
   END ;
   RESET ;    (* There should really be no RESET after KILL     *)
   ...        (* (actor might get suspended before its death).  *)
```

The acts would have to be modified to make them handle KILL exceptions appropriately. For example, the eternal outer loop would change into:

```
REPEAT
...
UNTIL  Reveal = KILL ;
```

Obviously, an actor with *one* outport must die as soon as a KILL request arrives. Correspondingly, a COPY node dies after *each* outport has got a KILL. But in *certain* cycles, the entire COPY node actor should die even when only *some* outports have got a KILL. Consider for example the simple cycle:




```
a    =    . . .     ;
b    =  a FBY b     ;
. . . = b           ;
```

same effect as: b = FIRST a

This COPY depends on itself. According to the simple rule, the right COPY outport will never get a KILL request, and the COPY will therefore never die. We must take a more global approach: we must view the subnet (consisting of COPY, FBY, and left FBY operand) as an *entity*, with the left COPY outport as the subnet outport. The rule would then be: "the subnet dies once each of its outports has got a KILL request." — One might be tempted into using a "trick", using a modified COPY which dies upon a *single* KILL request on one outport. But such a COPY would be useless in a slightly more complicated subnet (a combined vote of all subnet outports is needed, ↑ end of 4.7.2):

```
a    =   . . .     ;
b    =   . . .     ;
c    = a FBY d     ;
d    = b FBY c     ;
. . . = c          ;
. . . = d          ;
```

It must be clear by now why we printed K for ADVANCE, finalindex in the state transition tables and logs. Indeed, ADVANCE, finalindex can be substituted by KILL in our universal protocol, the difference lies outside the message passing behaviour.

6.4 Tailor made $\boxed{\text{COPY}}$ acts

The $\boxed{\text{COPY}}$ act offers many chances for optimisation: most applications do not need the generality of our universal $\boxed{\text{COPY}}$ act (↑ 4.6), and such restrictions can often be traded in for reductions in administration. Our $\boxed{\text{COPY}}$ act is very liberal in two respects:

- it imposes no maximum queue length,
- the relative "timing" between the different outports is unrestricted (i.e. index offsets between outports, and which outport is driving).

Sections 5.6 and 6.1 presented program analysis techniques for either property. This section provides shortcuts mainly for those cases where a maximum queue length is known. Our list of techniques is far from complete. – For the remainder of this section we use n to denote the number of $\boxed{\text{COPY}}$ outports.

Cyclic buffers are generally used when a maximum queue length $\boxed{q}$ is known. Such a buffer consists of an array $\boxed{v}$ of length $\boxed{q}$, a pointer $\boxed{\text{put}}$ which remembers where it last wrote into the array, and pointers $\boxed{\text{get}[i]}$ which remember where to read the array ($\boxed{\text{get}[i]}$ is dedicated to $\boxed{\text{COPY}}$ outport i, $i = 1 \dots n$). The general idea is then:

```
CONST
  ql    = 4711    ;    (* Maximum queue length.      *)
  ql1   = ql-1    ;
  n     = 3       ;    (* Number of COPY outports.  *)
VAR
  ptemp : ANYTYPE ;    (* The value to be put.      *)
  gtemp : ANYTYPE ;    (* The obtained value.       *)
  v     : ARRAY [0..ql1] OF ANYTYPE ;  (* Buffer     *)
  get   : ARRAY [1..n]   OF INTEGER ;
  put, i, j, k : INTEGER ;

BEGIN
  put       := -1           ;
  FOR    j := 1 TO n DO get[j] := 0 ;
  REPEAT
    ...
    (* Putting data into buffer:                    *)
    ptemp  := ...           ;
    put    := put + 1       ;
    FOR  j := 1 TO n
    DO  IF  get[j] + ql1 < put  THEN report_error ;
    k      := put MOD ql ;      (* wrap-around      *)
    v[k]   := ptemp         ;
    ...
    (* Assume outport i is not driving:             *)
    (* Getting data out of buffer:                  *)
    IF  put < get[i]  THEN report_error ;
    j      := get[i] MOD ql ;  (* wrap-around      *)
    gtemp  := v[j]             ;
    ...    := gtemp            ;
    get[i] := get[i] + 1       ;  (* ADVANCE       *)
    ...
  UNTIL FALSE ;
END ;
```

(The division remainder MOD helps to achieve a **wrap-around** effect: once the buffering has reached the end of array v it "jumps" back to the beginning.) This code can be simplified a good deal in specific cases:

- If $\boxed{q}$ = 2, the buffer consists only of $\boxed{v[0]}$ and $\boxed{v[1]}$. All the pointers toggle merely between 0 and 1:

```
put := 1 ;      get := 0 ;
  ...
  (* Putting data into buffer:      *)
  put      := 1-put   ;
  v[put]   := ptemp   ;
  ...
  (* Retrieving data from buffer:  *)
  gtemp    := v[get] ;
  get      := 1-get   ;    (* ADVANCE *)
  ...
```

- In a two-outport $\boxed{\text{COPY}}$, where the non-driving outport ***always*** lags two datons behind the driving one, we can even do without pointers altogether (***swapping*** buffer):

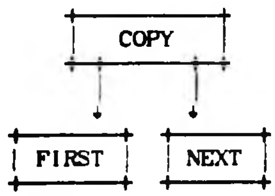
```
VAR  v0, v1 : ANYTYPE ;
  ...
  (* Putting data into buffer:      *)
  v1      := v0      ;
  v0      := ptemp   ;
  ...
  (* Retrieving data from buffer:  *)
  gtemp   := v1      ;
  ...
```

- Only ***one*** buffer cell $\boxed{v}$ is needed in a two-outport $\boxed{\text{COPY}}$ if the non-driving outport lags only ***one*** daton behind the driving one (↑ ***Version 4*** in 6.2):

```
VAR  v : ANYTYPE ;
  ...
  (* Putting data into buffer:      *)
  v       := ptemp  ;
  ...
  (* Retrieving data from buffer:  *)
  gtemp   := v      ;
  ...
```

- On the other hand, if the maximum queue length is known and if the ***entire history*** must be preserved (as in some versions of the $\boxed{\text{Sieve}}$ UDF), an ***array*** is most appropriate as buffer (just take the cyclic buffer and remove its wrap–around). Arrays are appropriate even if the queue length is ***unbounded***: it is best in that case to subdivide the available storage space into arrays according to the ***growth rate*** of the respective queues. The program ***collapses*** anyway once the buffer space is exhausted.

- A two–outport $\boxed{\text{COPY}}$ can be implemented altogether without a queue, as long as either outport disclaims the daton value early enough. Assume, the $\boxed{\text{COPY}}$ outports **o** and **O** progress in such a way that **O** gets a bare $\boxed{\text{ADVANCE}}$ always ***before*** **o** is requested $\boxed{\text{COMPUTE}}$ for the same daton index. The role of the outports may be swapped after each episode. This situation can arise if a variable **x** has two references of the kind:



or

```
p  =  FIRST x
q  =  NEXT  x

h  =  IF ... THEN x / 2
             ELSE x + 9   FI
```

6.5 Tagged Data Flow

Our $\boxed{\text{COPY}}$ act (↑ 4.6) is ***restrictive*** in one respect: it handles datons only in the sequence of ***increasing*** index (i.e. ***monotonically***, ↑ end of 3.1.2). This restriction is commonly made in Data Flow. We noticed, however, that acts without memory permit requests for datons in any sequence (↑ 4.5.7). A technique named "**Tagged Data Flow**" permits such ***random index*** computations. It is moderately difficult to change our implementation into tagged Data Flow; a redesign is required mainly for the actors with ***memory***: $\boxed{\text{COPY}}$, $\boxed{\text{READ}}$, $\boxed{\text{WRITE}}$ and UDFs.

In tagged DF, all $\boxed{\text{COPY}}$ node actors share one "***daton pool***" (faintly resembling a data base). Whenever a daton arrives at a $\boxed{\text{COPY}}$ inport, a "**bucket**" (a data record) is deposited in the pool, stating the value and the identity of the daton. The ***name*** of the $\boxed{\text{COPY}}$ inport actor can serve as identity tag. Whenever a $\boxed{\text{COPY}}$ outport gets a $\boxed{\text{COMPUTE}}$ request, it searches first the pool for the daton in question (using the daton index and $\boxed{\text{COPY}}$ inport name as search keys). If the search fails, the $\boxed{\text{COPY}}$ instructs its operand to determine the daton value. At suitable moments, the daton pool is cleared of defunct datons; reference counts or statistical methods (the "***retirement scheme***" [FaW83]) are used to identify defunctness. Tagged $\boxed{\text{READ}}$ works quite like tagged $\boxed{\text{COPY}}$, except that its datons remain permanently in the daton pool.

However, the tagged implementation becomes much more complicated once we allow ***recursive UDFs***. While a node actor is trying to evaluate one daton of a history, the system must be able to create another actor which evaluates another daton of the ***same*** history. Such a multi–level action is occasionally required for evaluating recursive Lucid definitions. All tagged DF implementations of Lucid use therefore a technique rather different from the one described in this thesis. Each of their node actors computes only a single daton, and dies then. The resulting high rate of actor creation and termination can be partly compensated by highly optimising the actor creation.

Generating good equivalent imperative code for tagged DF is very hard. The $\boxed{\text{WRITE}}$ act and our protocol can remain essentially unaltered. Only a UDF nesting control needs to be added; Ostrum/Wadge call this the "***place tag***". – Ostrum's Lucid interpreter [Ost81] is based on tagged Data Flow; it stores even all ***intermediary*** results (i.e. not only the $\boxed{\text{COPY}}$ queues) due to a present lack of program analysis. Denbaum's thesis [Den83] demonstrates how to compile tagged DF for a subset of Lucid, but with rather unsatisfactory code as result.

Why is the chapter on ***efficiency*** the place to discuss tagged Data Flow? The daton evaluation out of "***despair***" (↑ 4.6.5 and 5.6) can be completely avoided in tagged DF: its daton evaluation is free to skip index values since it can always come back to them. Tagged DF handles this situation clearly most efficiently. Pipeline DF excels in the ***simplicity of daton access***, where tagged DF needs an associative memory search. Moreover, the discarding of supposedly defunct datons occasionally forces tagged DF to re–evaluate datons.

6.6 Code Optimisation

There is a virtually unfathomable "box of tricks" for improving the efficiency of the generated code even further; quite important ones have already been presented earlier in this chapter. Here are three further tricks (in reverse order of difficulty):

Concurrent IF

It is easy to refine the IF operator so that it does not evaluate the ***condition*** operand **c** if the THEN operand **x** and the ELSE operand **y** deliver equal values anyway. Instances of:

IF c THEN x ELSE y FI

are simply substituted by:

IF c **OR** (x=y) THEN x ELSE y FI.

In general, this concurrent OR performs very poorly on von Neumann mono–processors, and it performs best if **c** is much more difficult to evaluate than **x** and **y**.

IF with Computed Constants

Recursive UDFs, in particular, tend to contain expressions like:

```
IF  FIRST expression  THEN x ELSE y FI
IF     index < t      THEN x ELSE y FI   // with constant t
```

The $\boxed{\text{IF}}$ switches in both cases, from a certain index on, to either choice; a re–evaluation of the condition will be unnecessary from then on. At that point can a ***net simplification*** ($\boxed{\text{KILL}}$ requests, operand redirection) be applied to the unsuccessful operand. The "arms" of the $\boxed{\text{IF}}$ do often contain a UDF recursion. Such a net simplification may prevent a UDF from inflating beyond all bounds.

Tail Recursion for Lucid UDFs

Recursive UDFs correspond to ***infinite*** nets (↑ 2.2), and the storage requirements of recursive UDFs ***increase*** whenever a new UDF is invoked. It is, however, occasionally possible to formulate acts for recursive UDFs so that they use **tail recursion** (or something resembling it), and they can lose their progressive storage requirements in this way.

Let $\boxed{X}$ be an actor for a recursive UDF, and let $\boxed{Y_0}$, ... $\boxed{Y_n}$ be the operand actors of $\boxed{X}$. The optimisation is only possible if all the ***actual*** operands $\boxed{Y_i}$ in the ***recursion*** of $\boxed{X}$ are particularly simple, i.e. if they are either identical to certain formal operands of $\boxed{X}$, if they are $\boxed{\text{COPY}}$ nodes, or if they deliver invariants (***constant*** or $\boxed{\text{FIRST} \ldots}$). They may even, and this is the most complicated case, deliver a formal operand **p** of $\boxed{X}$ with a ***simple modification*** (namely: **p** ***multiplied*** with an invariant, **p** with an invariant ***added***, ***ORed*** or ***ANDed***, or ***index*** of **p** with an invariant added). We exploit the fact that the effects of such operations can be ***accumulated*** in one storage cell.

This transformation generates a ***new UDF*** from the given UDF, so that the new UDF can do all the work of the given UDF, though ***without*** the growth in store. Further to the transformation of the ***actual operands*** and of the ***result***, above, a ***subnet transformation*** may have to be carried out. The subnet transformation is done as follows (before the translation): starting from the subnet outport we move upstream and mark every node of $\boxed{X}$ (including those in operand subnets) which contributes to the computation of the "current" daton with an index offset greater or equal zero. This marking requires that ***inner UDFs*** be expanded, in the worst case as often as there are $\boxed{\text{NEXT}}$ nodes in the UDF. The marking stops when each node of $\boxed{X}$, ***ignoring*** the invocation level, has been marked at least once (the transformation ***fails*** if a node needs to be marked more than once). The new UDF is then written so that it contains all the marked nodes, crossing invocation levels wherever needed. — The full description of the transformation will be the subject of a future paper. A recursive UDF may be expanded (↑ 6.2) once it has been transformed in this way.

Example ($\boxed{\text{Act\_Upon\_}}$)

It depends on the ***right*** $\boxed{\text{UPON}}$ operand value, how the operands (of the "current" activation) are transformed into the operands (of the "inner" activation). The result of the inner activation is transformed into invariants. $\boxed{\text{UPON}}$ was originally defined as the UDF:

```
UPON (a, k)  =  a  FBY  UPON ( p, NEXT k )
                WHERE  p =  IF (FIRST k) THEN  NEXT a
                                         ELSE       a  FI
                END ;
```

The transformation yields a new UDF:

```
NEWPON (a, k) = VALOF
                  d      = NEXT k                      ;
                  b      = IF  FIRST d  THEN NEXT a
                                        ELSE      a FI ;
                  result = b FBY NEWPON (b, d)         ;
                END ;
so that:

UPON (x, y)  =  x FBY NEWPON (x, 0 FBY y)
```

$\boxed{\text{NEWPON}}$ contains a tail recursion, and only $\boxed{\text{NEXT}}$ operations have to be accumulated. The resulting (non–recursive) code for $\boxed{\text{NEWPON}}$ can be merged with the $\boxed{\text{UPON}}$ adaptation into a reasonably short piece of code (it would be hard to explain the entire translation):

```
ACT Act_Upon_ ;
    LABEL 1 ;
    VAR
       superior, p0, p1  : ACTOR   ; request       : MSGTYPE ;
       index, count, now : INTEGER ; condi, empty  : BOOLEAN ;
       result            : ANYTYPE ;
    BEGIN
      ( . . p0, p1) := RECEIVE FROM (Creator) ;
      count := 0 ; now := -1 ; condi := TRUE ; empty := TRUE ;

    REPEAT
      WHILE TRUE DO BEGIN                                          :1
        (superior, request, index) := RECEIVE () ;

        WHILE  now < index      (* Catching up:              *)
        DO BEGIN                                                   :1
             IF   0 <= now
             THEN BEGIN                                            :1
                    condi :=  GetDaton (now+1,     p1) ;
                    EXCEPTION (ADVANCE, now+2) TO (p1) ;
                  END ;
             now := now + 1 ;

             IF   condi
             THEN BEGIN count := count + 1 ;
                        IF   empty
                        THEN EXCEPTION (ADVANCE, count) TO (p0)
                        ELSE empty := TRUE ;
           END    END ;

        IF   empty      (* Reluctant evaluation.              *)
        THEN BEGIN                                                 :1
               result := GetDaton (count,       p0) ;
               EXCEPTION (ADVANCE, count+1) TO (p0) ;
               empty  := FALSE ;
             END ;
                                                                   :1
        SEND (DATON, result) TO (superior) ;
      END ;             (* End of inner eternal loop.         *)

1:    (request, index) := Reveal ;        (* Exception part.  *)
      IF    (request = ADVANCE    )   AND
            (index   = finalindex)
      THEN  EXCEPTION (request, index) TO (p0, p1) ;
      RESET ;
    UNTIL FALSE ;       (* End of outer eternal loop.         *)
    END ;               (* End of Act_Upon_ .                 *)
```

The example shows a further application of the "catching up" mechanism (↑ 6.2, Version 4), it uses the $\boxed{\text{FIRST}}$ / $\boxed{\text{NEXT}}$ optimisation of $\boxed{\text{COPY}}$ (for the variable $\boxed{\text{d}}$), the invariant $\boxed{\text{IF}}$, and UDF tail recursion with accumulation of $\boxed{\text{NEXT}}$. The $\boxed{\text{UPON}}$ actors do not build up internal queues. – Similar methods are applied to obtain the WVR act:

Example (Act_Wvr_)

```
ACT Act_Wvr_ ;
    LABEL 1 ;
    VAR
       superior, p0, p1 : ACTOR   ; request      : MSGTYPE ;
       result           : ANYTYPE ; condi, empty : BOOLEAN ;
       index, count, lcount, now : INTEGER ;
    BEGIN
      count := 0 ; lcount := 0 ; now := -1 ; empty := TRUE ;
      ( , , p0, p1) := RECEIVE FROM (Creator) ;

    REPEAT
      WHILE TRUE DO BEGIN                                              :1
        (superior, request, index) := RECEIVE () ;

        WHILE  now < index DO
          BEGIN REPEAT                                                 :1
                  IF   lcount < count
                  THEN EXCEPTION (ADVANCE, count) TO (p0) ;
                                                                       :1
                  condi := GetDaton (     count,     p1) ;
                  count := count + 1 ;
                  EXCEPTION      (ADVANCE, count) TO (p1) ;
                UNTIL condi ;
                now      := now   + 1 ;
          END ;
        lcount           := count - 1 ;

        IF   empty       (* Reluctant evaluation.                *)
        THEN BEGIN                                                     :1
               result := GetDaton (        lcount,     p0) ;
               lcount := lcount + 1 ;       empty := FALSE ;
               EXCEPTION          (ADVANCE, lcount) TO (p0) ;
             END ;
                                                                       :1
        SEND (DATON, result) TO (superior) ;
      END ;              (* End of inner eternal loop.           *)

1:    (request, index) := Reveal ;        (* Exception part.  *)
      IF             request = ADVANCE     THEN
        BEGIN IF   index   = finalindex
              THEN EXCEPTION (request, index) TO (p0, p1)
              ELSE IF  empty  THEN
                     BEGIN lcount := lcount + 1 ,
                           EXCEPTION (request, lcount) TO (p0) ;
                     END ;
              empty := TRUE ,
        END ;
      RESET ;
    UNTIL FALSE ;        (* End of outer eternal loop.           *)
    END ;                (* End of Act_Wvr_ .                    *)
```

6.7 Discussion

The purpose of this chapter was to destroy the myth that Lucid programs are inherently inefficient. It gave only an idea of possible optimisation techniques. The chapter has been somewhat vague concerning when and how to apply each optimisation, it has been merely a fairly unsystematic collection of "tricks". A closed and comprehensive theory of optimisation would be desirable, and such work is under way in a number of places. – Most of the optimisations techniques in this chapter were aimed at a von Neumann mono–processor. If we applied them to our $\boxed{\text{Sieve}}$ program we would end up with a single actor, created from the following act:

Example ($\boxed{\text{Sieve}}$): final result

```
ACT Act_Primes_ ;
    LABEL 1, 2 ;
    VAR
      index, result, t, i          : INTEGER ,
      primes : ARRAY [1..2000] OF INTEGER ;
    BEGIN
      index := 0 ;       t := 0 ;

      REPEAT
        result := index + 2 ,   (* † 6.2, Version 6.  *)

2:      FOR i := 1 TO t
        DO  IF  (result MOD primes[i]) = 0  THEN GOTO 1 ;

        WRITE (result) ; t:=t+1 ; primes[t] := result ,
        GOTO 2 ;

1:      index  := index + 1 ;
      UNTIL t = 2000 ;      (* End of eternal loop  *)
    END ;
```

But what is the $\boxed{\text{GOTO 2}}$ doing there? The program would only gain if that instruction was ***omitted***. – This is a very interesting point. The translation of the Lucid program really yields the program as shown, with the $\boxed{\text{GOTO 2}}$ in it, though the Lucid program is easily corrected. Is the Lucid program meant to specify the operations which shall be carried out, or is it just a mathematical definition of the result history? There is no universally accepted answer to this question. One might give the Lucid compiler an option stating the approach favoured by the user. (The former view might be most suited during program development.)

CHAPTER VII: Areas of Further Research

7.0 Introduction

Quite a few aspects of implementing Lucid have been omitted in this thesis. This omission was sometimes deliberate, sometimes not. Some explanations would have distracted from the true issue of the thesis, they would have overloaded the thesis. For some topics, simply too little is presently known, so that answers could not be based on well founded knowledge. Some areas where further research is indicated have already been mentioned in the pertaining chapters:

- Obviously, the next action now due is the implementation, on real machines, of the essence of this thesis. A working system is always the most credible demonstration of success. Quite commonly, such a system sparks off a wealth of new ideas; the use of our pLucid system [FMY83] has very much had this effect. Only the most essential parts of this thesis have so far been implemented, since it was felt that an emphasis should be put on careful planning and on scientific analysis.
- Scheduling strategies need to be developed (a) for a revised Lucid with more than one WRITE, and (b) for running Lucid on a multiprocessor network. Ideally, an operating system should be developed which takes into account the demand driven and potentially concurrent nature of Lucid.
- The efficiency of the Lucid system can be improved by protocol extensions, by the provision of further highly adapted acts, and by further program analysis methods. Provisions for actor termination fall also into this category. The long term aim is clearly the development of a systematic and comprehensive theory of optimisation, superseding the present patchy approach.
- The specific advantages of tagged DF and pipeline DF have been contrasted (↑ 6.5). Lucid programs with reverse dependencies are not pipeline computable

without major rewriting. Is there a general algorithm for making all Lucid programs monotonic, so they can run in pipeline DF?

7.1 Other Operational Models

In our translation, the underlying execution strategy has been demand driven DF with pipelines as buffers. Chapter I gave the reasons for this particular choice. However, there are situations where one of the other strategies would be more appropriate.

Lucid implementations have been done for the Manchester Data Flow machine [Bus79, Sar82]; that machine is truely data driven and leans in a direction rather opposite to the one taken by this thesis. Our translation generates very efficiency conscious code: an evaluation is initiated only when its result is needed. However, generosity can suit even a miser: some premature evaluations are cheaper than the administration for their delay. We should therefore investigate where data drive would improve our code.

Especially our WRITE act (↑ 4.5.4) reflects the data driven and pipeline oriented nature of the operating system. However, a demand driven system (like pLucid) comes really into its own when put together with other demand driven systems, such as data base query systems. A demand driven operating system exists already, as an academic exercise, but the relevance of this topic has not been fully appreciated, yet.

7.2 Language Extensions for Lucid

Even though Lucid is already highly developed, various extensions would make it even more usable: arrays, types, higher order functions (functions operating on functions), and time dependent functions. Many extensions are a mere question of sweat, but time dependent functions ask for a major re–think of Lucid altogether, including its implementation technique:

Pure Data Flow is a restriction of Data Flow under which only *functional* operators are permitted. An operator is **functional** if its result is entirely determined by the values of its operands. An operator whose result depends on the "wall clock" *time of execution* is clearly *non*–functional. We have so far only bothered about Lucid as a *functional* programming language (↑ chapter II), i.e. the version of Lucid where all the operators *are* functional. Lucid has originally been designed to be a functional language, and an interface to the *operational* domain is bound to produce problems.

There are a few situations which require non–functional means; for example, the operating system must be able to test whether the user has struck a key, or to ask for the time of the day. One might simply try to enrich Lucid by new functions $\boxed{\text{Buffer\_Full}}$ and $\boxed{\text{Time\_Now}}$. This approach is inappropriate in many situations. It may, in tagged Data Flow in particular, lead to the queuing a vast numbers of irrelevant data. Wadge suggested another method by introducing **hiatons** (the Greek word *hiatus* means "pause"), special data items indicating "*no daton available*". The use of hiatons makes a total redesign of the Lucid system necessary, even the language itself may need a few extensions. Hiatons can occur anywhere in a history, they don't occupy daton positions in the history, and it is therefore possible to *filter* all the hiatons out of the history (to "*de–hiatonise the history*"). Hiatons have implications on many aspects of Lucid, and further research is needed before conclusive answers can be given.

Summary

The thesis has described a complete implementation method for Lucid, based on Message Passing. The description has been presented step by step, starting with a "conditioning" stage, followed by the main translation, and ending with code optimisation. All the essential items of code are readily contained in the text. The thesis can thus be used directly as a guide for the implementation on any computer system with Message Passing. Due to its modularity, universal components can be easily replaced by optimised ones. The modularity makes it also easy to check the correctness of every stage. The correct execution has been illustrated by special diagrams, execution logs, which highlight particularly the sequence of events in the case of concurrent execution.

Bibliography

AsW76 — Ashcroft E.A. & Wadge W.W., "LUCID, a Formal System for Writing and Proving Programs", Theory of Computation Report No.4, 35 pages, Univ.of Warwick, January 1976

AsW77a — Ashcroft E.A. & Wadge W.W., "LUCID, a Nonprocedural Language with Iteration", p.519...526, CACM 20, No.7, July 1977

AsW77b — Ashcroft E.A. & Wadge W.W., "Scope structures and defined functions in LUCID", Theory of Computation Report No. 21, 41 pages, Univ.of Warwick, October 1977

AsW79a — Ashcroft E.A. & Wadge W.W., "A Logical Programming Language", Report No CS—79—20, 57 pages, Univ.of Waterloo/Canada, June 1979, (revised March 1980)

AsW80 — Ashcroft E.A. & Wadge W.W., "Structured LUCID", Theory of Computation Report No. 33, 54 pages, Univ.of Warwick, March 1980 also: report CS—79—21, Comp.Sci., Univ.of Waterloo

AsW83 — Ashcroft E.A. & Wadge W.W., "Lucid, the Dataflow Programming Language", Academic Press, 300 pages approx, to be published 1983,

BrH75 — Brinch Hansen P., "Concurrent Pascal Report", California Inst. of Technology, June 1975

Bus79 — Bush V.J., "A Data Flow Implementation of Lucid", M.Sc. dissertation, Univ.of Manchester, October 1979

Car76 — Cargill T.A., "Deterministic Operational Semantics for Lucid", Report CS—76—10, Univ.of Waterloo, June 1976

Den83 — Denbaum C., "A Demand—Driven Coroutine—Based Implementation of a Non—Procedural Language", Ph.D. thesis, 231 pages, Univ of Iowa, May 1983

Dij75 Dijkstra E.W.,
"Guarded Commands, Nondeterminacy and Formal Derivation of Programs",
p.453...457, CACM, Vol. 18, No. 8 (August 1975)

DMN68 Dahl O.-J., Myhrhaug B., Nygaard K.,
"The SIMULA 67 Common Base Language",
Norwegian Comp. Center, Oslo 1968.

Fab68 Fabry R.S., "Preliminary description of a supervisor for a computer organised around capabilities",
Quart. Prog. Report No 18, Sect. II A,
Inst.Comp.Res., Univ. Chicago

Far77 Farah M.,
"Correct Compilation of a Useful Subset of LUCID",
159 pages, Ph.D. Thesis, Waterloo, Ontario/CAN 1977

Far80 Farah M., "Correctness of a Lucid Interpreter Based on Linked Forst Manipulation Systems",
p.3...26, Intern. J. Computer Math., Section A8, 1980

Fau82 Faustini A.A.,
"The Equivalence of an Operational and a Denotational Semantics for Pure Dataflow",
Ph.D. thesis, 191 pages, Univ.of Warwick, March 1982

FaW83 Faustini A.A. & Wadge W.W.,
"The pLucid Interpreter and its Retirement Scheme",
talk at "Swedish Workshop", Brighton, June 1983,
Report (in preparation)

Fin81 Finch B., "An Operational View of Lucid",
Research Report CS-81-37, Computer Science,
Univ.of Waterloo/Canada, 33 pages, December 1981,

FMY83 Faustini A.A., Matthews S.G. & Yaghi A.AG.,
"The pLucid Programming Manual",
Distributed Computing Project Report No. 4,
60 pages, Univ.of Warwick, April 1983

FrW76 Friedman D.P. & Wise D.S.,
"CONS should not evaluate its arguments",
in: Michaelson S. & Milner R. (Eds.), p.257-284,
3rd Int'l Coll. on Automata, Languages and Prog.,
Edinburgh Univ. Press, Edinb. 1976

Gar78 Gardin P., "Une implementation de Lucid", M.Sc. Thesis, 89 pages, Notre-Dame University Namur/Belgium, 1978

HBS77 Hewitt C., Bishop P. & Steiger R., "A Universal Modular ACTOR Formalism", Proceedings, Formalisms for AI, Session 8, p.235...245, probably 1973

HeM76 Henderson P. & Morris J., "A Lazy Evaluator", Record 3rd SIGPLAN-SIGACT Symp. on Principles of Prog. Lang. (Atlanta, Jan 1976), p.95-118, ACM, New York 1976

Hoa74 Hoare C.A.R., "Monitors, an Operating System Structuring Concept", p.549...557, CACM, Vol.12, No.10 (October 1974)

Hoa78 Hoare C.A.R., "Communicating Sequential Processes", p.666...677, CACM, Vol.21, No.8 (August 1978)

Hof78 Hoffmann C.M., "Design and Correctness of a Compiler for a Nonprocedural Language", p.217...241, Acta Inf. 9, 3 (1978)

Inm82 INMOS, "OCCAM", (company publication), 45 pages, INMOS Ltd., Whitefriars, Lewins Mead, Bristol BS1 2NP, U.K. (1982)

Koc80 Koch A., "MENYMA, Design and Impl. of a Message Oriented Language", Master's Thesis, Computer Science, Univ.of Waterloo/Canada, December 1980,

Lam78 Lamport L., "Time, Clocks, and the Ordering of Events in a Distributed System", p.558...565, CACM Vol.21, No.7 (July 1978)

Lan66 Landin P.J., "The Next 700 Programming Languages", p.157...166, CACM Vol.9, No.3 (March 1966)

Lis74 Liskov B.H., "A Note on CLU", Computation Structures Group Memo. 112, M.I.T, Cambridge, Mass., 1974

MaT79 May M.D. & Taylor R.J.B., "The EPL programming manual", Distributed Computing Project Report No. 1, 18 pages, Univ.of Warwick, May 1979

Ost81 Ostrum C.B.,
"LUTHID 0.0 Preliminary Reference Manual and Report",
Dept. of Computer Science, University of Waterloo,
22 pages, Waterloo/Ontario, Canada, 1981

Sar82 Sargeant J., "Implementation of
Structured Lucid on a Data Flow Computer",
M.Sc. dissertation, Univ.of Manchester, October 1982

Wad79 Wadge W.W., "An Extensional Treatment of Dataflow Deadlock",
Theory of Computation Report No. 28,
Univ.of Warwick, April 1979, 15 pages
also: Springer Notes on Comp. Science #70, p.285...299,
(Conference on "Semantics of Concurrent Computation",
Evian/France, July 1979)

Wen80 Wendelborn A.L.,
"The Implementation of a Simple Non-Procedural Language",
TR 80-4, 26 pages, Univ.of Adelaide, 1980

Wen81 Wendelborn A.L.,
"Implementing a Lucid-like Programmming Language",
Australian Computer Science Communications, Vol 3, No 1,
p.211-221, May 1981

Wen82 Wendelborn A.L.,
"A Data Flow Implementation of a Lucid-like Prog. Language",
TR 82-06, 36 pages, Univ.of Adelaide, 1982

Wir71 Wirth N., "The Programming Language PASCAL"
Acta Informatica 1, 35-63-1971

Wir69 Wirth N., "How to Live without Interrupts" or similar,
p.489...498, CACM Vol. 12 No 9 (Sept. 1969)

Wir75 Wirth N., "MODULA, a Language for Modular Programming",
Bericht 18, Inst. fuer Informatik, ETH, CH-8092 Zuerich,
Zuerich 1975 (?)

Wil72 Wilkes M.V., "Time-sharing Computer Systems",
2nd edition, Macdonald/Elsevier London 1972

WuS73 Wulf W. & Shaw M., "Global Variable Considered Harmful",
Carnegie-Mellon Univ., Pittsburgh PA, SIGPLAN Bulletin 1973

Yag83 Yaghi A.AG.,
"An Intensional Impl. Technique for Functional Languages",
personal communication, Ph.D. thesis (in preparation),
Dept. of Computer Science, Univ.of Warwick,

Appendix A: The BNF of Lucid

Here is the BNF of Lucid, the way it is used throughout this thesis. This is a subset of the language pLucid [FMY83]. The algebra of pLucid comprises lists (as in POP–2 or LISP) and the pertaining operators. Lists are a completely separate topic area; they have been omitted in this thesis for the sake of clarity, but they can be added any time without necessitating a revision of the thesis. We go even further, we use a minimal algebra which comprises only TRUE, FALSE, ERROR and all the integers. Examples may occur in this thesis which exceed this minimal algebra (using real 3.14159 or a string like "Hello there"); the reader is asked to take the BNF as suitably extended.

In the BNF formalism we use the following notation:

< > — every ***meta term*** is enclosed in angle brackets,

::= — reads as ***<meta term> is defined by <meta expression>***,

| — reads as ***<meta expression> or <meta expression>***,

{ } — denotes ***possible repetition zero or more times*** of the enclosed ***<meta expression>***,

// — precedes comments.

The Lucid syntax is defined by the following BNF:

```
<program>        ::=  <expression>

<expression>     ::=  <primary>
                    | <prefix operator> <primary>
                    | <primary> <infix operator> <primary>
                    | <where clause>

<primary>        ::=  <constant>
                    | <vari>
                    | <if expression>
                    | <function ref>
                    | ( <expression> )
                    |   <expression>   // precedence permitting
```

```
<constant>          ::=  <numeric constant>
                       | TRUE | FALSE | ERROR

<numeric constant>::=  <digit> { <digit> }
                       | - <numeric constant>

<digit>             ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<letter> ::=  A | B | C | D | E | F | G | H | I | J | K | L | M
            | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
            | a | b | c | d | e | f | g | h | i | j | k | l | m
            | n | o | p | q | r | s | t | u | v | w | x | y | z

<alphanumeric>      ::=  <digit> | <letter>

<identifier>        ::=  <letter> { <alphanumeric> }

<vari>              ::=  <identifier>       // name of a variable

<prefix operator> ::=  -  | NOT | FIRST | NEXT

<infix operator>    ::=  +   | -   | *   | /   | MOD
                       | LE  | LT  | GT  | GT  | NE  | EQ
                       | <=  | <   | >   | >=
                       | AND | OR
                       | FBY | WVR | ASA | UPON

<if expression>     ::=  IF <expression> THEN <expression>
                                         ELSE <expression> FI

<where clause>      ::=  <expression> WHERE <body> END

<body>              ::=  { <currenting> }   { <definition> }

<currenting>        ::=  <vari> IS CURRENT <expression> ;

<definition>        ::=  <simple def>
                       | <function def>

<simple def>        ::=  <vari> = <definiens> ;

<function def>      ::=  <func> ( <formals> ) = <definiens> ;

<definiens>         ::=  <expression>

<func>              ::=  <identifier>         // function name

<formals>           ::=  <vari> { , <vari> }

<function ref>      ::=  <func> ( <actuals> )

<actuals>           ::=  <expression> { , <expression> }
```

Note: Throughout pLucid ~ is used instead of – .

<currenting> is described in appendix B.

Lucid programs can contain comments directed solely at the human reader: the compiler ignores double backslashes // and everything on their right hand side within the line.

The following identifiers are reserved as keywords:

AND ASA CURRENT ELSE END EQ ERROR FALSE FBY FI FIRST GE GT
IF IS LE LT MOD NE NEXT NOT OR THEN TRUE UPON WHERE WVR

(Throughout this thesis, keywords are written in capitals and variables in lower case. However, that rule is not part of real Lucid but intended to improve legibility)

Here is a short description of the operators of our algebra:

prefix op.	meaning
−	Arithmetic inverse, the operand * (-1) .
NOT	Boolean negation.
FIRST	Infinite extension of initial daton.
NEXT	Op history with initial daton removed.

infix op.	meaning (T = TRUE , F = FALSE)
+	Sum of the two operands.
−	Result of subtracting the right op from the left one.
*	Product of the two operands.
/	Quotient from dividing the left op by the right one without remainder (13 DIV 7 = 1 , (-13) DIV 7 = −1).
MOD	Remainder from dividing the left op by the right one.
AND	T if both ops are T, F otherwise.
OR	F if both ops are F, T otherwise.
GT >	T if left op Greater Than right op, F otherwise.
GE >=	T if left op Greater or Equal right op, F otherwise.
LT <	T if left op Less Than right op, F otherwise.
LE <=	T if left op Less or Equal right op, F otherwise.
EQ	T if left op EQual to right op, F otherwise.
NE	T if left op Not Equal to right op, F otherwise.
FBY	Initial daton of left hist prepended to right hist.
UPON	Repeats left daton while right daton is FALSE.
WVR	Ignores left daton whenever right daton FALSE.
ASA	First left daton whose right daton is TRUE.

Impossible computations, like a division by zero, yield ERROR. This is a special value indicating "something went wrong in the computation of this daton". It is impossible to guarantee the indication of every error (halting problem!) .

The BNF defines expressions in terms of ***primaries***, which are merely particularly "well mannered" expressions. All primaries are therefore expressions. Just a variable or a constant is a primary. Enclosing an expression in brackets promotes it to a primary. The construct $\boxed{\text{IF c THEN x ELSE y FI}}$ is a primary, where $\boxed{c}$ $\boxed{x}$ and $\boxed{y}$ are expressions. Lastly, any function reference is also a primary. Any expression is either just a primary, or a primary with a prefix operator put in front, or two primaries with an infix operator between them.

The precedence rules and the association rules permit the omission of brackets in many cases. These rules have been detailed in section 2.1.2.

Appendix B: "Currenting", the Lucid Approach to Embedded Iteration

B.0 Introduction

It is generally accepted for imperative programming languages that *iteration* is the construct which increases their expressive power most decisively. Iteration comes to full fruition if it is embedded in some larger computation (*embedded iteration*). Incidentally, it is well known that imperative iteration can be simulated by recursion. In common computer jargon, iteration means *repetition*, and the term is commonly applied in two contexts: mathematical iteration (as in the Newton–Raphsen algorithm for $\boxed{\text{sqrt}}$) on the one hand, and multiple application on the other (like setting an array to zero). Both are *bulk computations* in a sense. The term iteration is, strictly speaking, not applicable to a non-imperative language like Lucid, but one would expect Lucid to comprise a denotational *counterpart* to iteration. Confusion can result from the fact that already a *single* Lucid assertion can represent a bulk computation, since it expresses a whole *stream* of data objects (due to the Lucid algebra).

In less operational terms, any substantial programming language must satisfy the following requirements:

(1) It should provide means for the definition (and application) of new operators. An *operator* is a generalised (abstracted) instruction, i.e. its actual operands are specified only in the application stage. A set of fundamental operators is usually pre-given. The definition of any new operator is achieved by abstractly stating the actions symbolised by the operator. An operator is *recursive* if it references itself (in its definition), and this includes any *indirect* self-reference. In a broad sense, every subprogram is among the operators, as is the body of any $\boxed{\text{DO}}$ loop or $\boxed{\text{WHERE}}$ clause. (According to our definition, the term "*operator*" includes function subprograms. The term "*function*" has a specific mathematical meaning which might interfere in this context.)

(2) Every programming language should provide a method for specifying the application of any operator to a *collection* of operands. (In ALGOL this may be an array or may be the successive values held in a storage cell, in Lucid this may be the datons of a history.) Such a **multiple application** may well produce a *combined result* (e.g. the computation of an average value within the collection).

(3) There should furthermore be a provision for taking the combined result of such a multiple application, and for delivering it as a single value to the *larger* computation (in which the multiple application is embedded).

Lucid satisfies requirement (1), the whole language is designed around operator definitions. Every Lucid assertion is an operator definition. A demand for the program's result is, operationally speaking, the cause for all computations. Requirement (2) is satisfied since every variable stands for a sequence of data objects. Future versions of Lucid which have arrays (and operators on arrays) offer a further method of satisfying this requirement. But at the this point in the discussion we do not seem to have anything fitting requirement (3).

A combined result of multiple operator application, requirement (2), can be formed by use of NEXT and FBY. Here is for example the running total of history X:

```
Sum  =  X + (0 FBY Sum) ;
```

Every daton value of Sum is based on an entire initial segment of X.

```
s99  =  Sum ASA index = 99 ;
```

means therefore that many computations are involved in the production of one result. The assertion for s99 has the drawback that it asserts just one *constant* value. There should be a way for executing numerous *low ranking* computations which, taken together, deliver a single *final result* daton (a bit like s99) to a *higher ranking* assertion. This should be followed by renewed low ranking computations which in turn produce the next result daton.

Iteration without a means for such ***embedding*** (i.e. without sub–computations) is of limited use. We shall see that Lucid achieves embedding in a rather natural way.

B.1 Structured Lucid

Early in the development of Lucid [AsW76], certain mathematical concepts were identified, and were then chosen as the foundations of the language. A suitable syntax was then worked out. The syntax has indeed been subject to refinement up to the present day. Valuable insights into the underlying concept can be gained from looking at the earlier development stages of Lucid, though only few traces bear witness in the present form.

Ashcroft and Wadge describe in their paper "Structured Lucid" [AsW80] how a technique called "***currenting***" equips Lucid with embedded iteration. They show how Lucid is conceptually derived from the languages USWIM [AsW79a] (which itself is a derivative of Landin's ISWIM [Lan66]) and Basic Lucid [AsW77a]. The language **ULU** is obtained when the USWIM structures (WHERE clauses and functions) are built on top of the Basic Lucid objects (infinite histories). On the other hand, putting Basic Lucid on top of USWIM yields the language **LUSWIM**. Both languages have exactly the same syntax. But they differ in semantics, in particular in the effect which structures (WHERE clauses) have on variables (histories). Lucid is an amalgamation of LUSWIM and ULU, and the divergence in semantics has been resolved by declaring each variable either as **currented** or **uncurrented**. (In [AsW80] a different terminology was used, and the ***typeface*** of each variable indicated its currenting status. However, this distinction by typeface proved rather impractical.)

It was later decided to consider any variable by default as uncurrented, and to state explicitly when the variable was meant to be currented instead. Uncurrented variables are the easiest to understand, since their entire histories are imported into WHERE clauses without any change. The declaration x IS CURRENT y in a WHERE clause

indicates that the new variable x is the currented version of the variable y, where y refers to a variable y defined outside the $\boxed{\text{WHERE}}$ clause. Currenting is occasionally also called "*freezing*", since the enclosing environment is held in an invariant state, as if it was frozen. While [AsW80] introduces currenting denotationally, we will use the operational point of view throughout our explanation, since this seems easier to understand.

It will be shown below that currenting can be expressed entirely in ULU terms, and consequently LUSWIM can be viewed as a special case of ULU. In other words, every Lucid program can be expressed in terms of ULU alone. (***Not*** all Lucid programs can be expressed in terms of LUSWIM alone.) Incidentally, ULU is essentially the language presented in chapter I.

B.2 Present Lucid

Global variables ("imported" variables) have been defined in the description of Lucid (↑ chapter I). Any global variable **y** can be ***currented*** by placing at the beginning of the $\boxed{\text{WHERE}}$ clause the declaration:

```
x IS CURRENT y ;
```

(The expression on the right (here: **y**) is evaluated in the environment which ***encloses*** the $\boxed{\text{WHERE}}$ clause, **x** and **y** can therefore even be identical identifiers.) The following assertion might occur in a program:

```
result  =  f(x, i)                 # f is an arbitrary function,
           WHERE                   # x is the currenting of
              x IS CURRENT y ;     # the global variable y,
           END ;                   # i is an uncurrented global.
```

fig. B1: an assertion with currenting

The variable x is the currented version of y, where y is a global variable. To function f, x will appear like a constant, all its components are equal. The history of y is mapped into a sequence of histories x, where the k–th subhistory x consists throughout of

components identical to y_k , where y_k is variable y at index k. The function f is applied individually to each (constant) subhistory, and consequently there is a sequence of result histories of function f. The result of the WHERE clause is the sequence of the n-th components of the application of f to the n-th subhistory x, with n ranging from 0 to infinity. (Note that, naturally, the computation restarts from index 0 for each single invocation of f.) The index progresses inside the WHERE clause thus in the following ***triangular pattern***:

```
0  1  2  3  4  5  6  7  8  9  -->  index of the global to be currented,
                                     = index outside the WHERE clause.

0  0  0  0  0  0  0  0  0  0 ...   |
|  1  1  1  1  1  1  1  1  1       |
|  |  2  2  2  2  2  2  2  2       |  indices for which
|  |  |  3  3  3  3  3  3  3       |  the subhistories
|  |  |  |  4  4  4  4  4  4       |  (inside the WHERE-clause)
|  |  |  |  |  5  5  5  5  5       |  are computed;
|  |  |  |  |  |  6  6  6  6       |  only the last value is taken.
|  |  |  |  |  |  |  7  7  7       |
|  |  |  |  |  |  |  |  8  8       v
|  |  |  |  |  |  |  |  |  9
|  |  |  |  |  |  |  |  |  |
v  v  v  v  v  v  v  v  v  v
0  1  2  3  4  5  6  7  8  9   index of the result
```

fig. B2: the triangular execution pattern

We have not yet mentioned the other operand of f, namely i. Each invocation of f gets the *entire* history of i, since i is *not* currented in any way. Because of functionality, it does not matter whether i is re-computed each time or whether i is computed once only with copies being given to each invocation of f. (Repeated evaluation of a function yields the same result as long as all operands remain identical.) The same would apply to any other uncurrented variable occurring in the WHERE clause. Below we will study another example program with a WHERE clause which contains both a currented and an uncurrented global variable.

It is particularly interesting to study an unusual WHERE clause which has currented as well as uncurrented global variables, but where none of the currented variables is actually used. Is the result really invariant to the addition of these superfluous variables? Instead of computing the result in a straight tour through the indices (0, 1, 2, ...) the currented variables enforce the repetitive triangular pattern of figure B2. Because of this considerably changed execution pattern some effect on the result would not come as a surprise. But since all operators are functions, and all operands are either local or uncurrented global variables the result is indeed invariant. It can not be distinguished whether any intermediary value has been computed anew, or whether a value from a previous computation has been re-used.

Structured Lucid allows even the currenting of (non-nullary) functions. This means effectively the currenting of all global variables which occur in the definition of that function. This currenting of functions has been abolished in the latest versions of Lucid, to keep matters simple. There is hardly any *useful* function where both versions (the currented and the uncurrented one) are equally needed. The currenting of the global variables can therefore be carried out inside the function definition itself, which is better style anyway (in the software engineering sense).

If we have another look at the figure above, it is evident that the "*daton production rate*" of the computation inside the WHERE clause is *greater* or equal the rate in the environment. In other words, we have some form of embedded iteration. No proof will be given here that currenting is a comprehensive technique for embedded iteration, or in other words, that point (3) is satisfied in *every* respect. One might even be led to believe that the triangular pattern (↑ fig. B2) restricts the range of application to those very few situations where the number of computations inside the WHERE clause grows exactly with *index+1*. However, this restriction can be overridden by enclosing the WHERE expression (*preceding* the keyword WHERE, for example *f(x,i)*, ↑ fig. B1) in an ASA with an appropriate *terminating condition*, like:

```
( f(x, i)  ASA  condition(...)  )
  WHERE ... IS CURRENT ... END
```

Since this expression contains the ASA operator, it may appear strange, at first glance, that this WHERE expression does *not* necessarily yield a *constant* history. The

$\boxed{\text{ASA}}$ lies inside a $\boxed{\text{WHERE}}$ expression with currenting, which means that only a single result daton is picked out. For each pass of this $\boxed{\text{WHERE}}$ clause, the $\boxed{\text{ASA}}$ expression is computed *anew* with fresh currented values, which may produce a totally different $\boxed{\text{ASA}}$ result in every pass.

One last remark. It has been described in chapter I that assertions can be freely moved into and out of $\boxed{\text{WHERE}}$ clauses as long as certain syntactic rules (identifier clashes) are not violated. Matters are different if a $\boxed{\text{WHERE}}$ clause has a global variable, and if that variable is *currented* in the $\boxed{\text{WHERE}}$ clause. In such a case the assertion for the variable can not in general be moved across the $\boxed{\text{WHERE}}$. This is possible only if the operators in that assertion commute with currenting. A discussion of this is found in [AsW80].

B.3 Currenting Expressed by Recursion

Can currenting be expressed purely by the means described in chapter I?

The triangle (↑ fig. B2) shows that the result history is constructed out of separate invocations of the function f, one for each result daton. The result is composed of the initial daton of the initial function invocation, followed by the daton at index 1 of the next invocation, followed by the daton at index 2 of the function invocation after that, etc. Regarding function parameters, each function invocation has *full* access to any uncurrented parameter. For *currented* parameters, on the other hand, the initial function invocation obtains a constant history which consists purely of copies of the initial daton of the parameter. The next invocation obtains the constant history generated from the daton at index 1, and so on.

Taken together, the same result as in fig. B1 would be computed by:

```
                                                              # index
result  =          f (                      FIRST x, i)  FBY   # 0
              NEXT f (            .    FIRST NEXT x, i)  FBY   # 1
         NEXT NEXT f (            FIRST NEXT NEXT x, i)  FBY   # 2
    NEXT NEXT NEXT f (       FIRST NEXT NEXT NEXT x, i)  FBY   # 3
NEXT NEXT NEXT NEXT f (  FIRST NEXT NEXT NEXT NEXT x, i)  FBY   # 4
NEXT NEXT NEXT NEXT NEXT f (FIRST NEXT NEXT NEXT NEXT NEXT x, i)  FBY   # 5
...
etc etc
```

This can be expressed by a recursive function. We call this function Igloo , since currenting has the effect of permitting *live* computations in a *frozen* environment. Obviously, nothing special needs to be done about the un-currented parameter i; it is passed untouched to each new invocation of f, and its history restarts therefore always right from the beginning. The currented parameter x is not difficult to express either. With each "round trip" of the recursion one more initial element is stripped off, the resulting history is made into a constant by the application of FIRST , and this is then passed to f as a parameter. The Igloo function must therefore have an appearance like:

```
Igloo (..., newx)  =  func (FIRST newx, i)
                      FBY  Igloo (..., NEXT newx);

result = Igloo (..., x) ;
```

Here, func is related to f, but it is *identical* to f only for the initial result daton. One further NEXT must be applied to f for each successive result daton, i.e. one per recursion of Igloo . One feels tempted to generate the new function, in each "round trip" of the recursion, by *composing* ("•") a NEXT with the old function; the starting "value" would be the plain function f. To do this, we would need a function parameter in Igloo , like:

```
Igloo (function, ...)  =  ...  FBY Igloo ( NEXT • function , ...) ;
result = Igloo (f, ...) ;
```

Sadly, function parameters are presently not allowed in Lucid. The multiple application of NEXT must therefore be simulated otherwise. But even that problem can be overcome. Remember that, for any constant *n*,

```
expression WVR (n = index)
```

has the same effect as applying $\boxed{\text{NEXT}}$ *n* times to the expression. The complete $\boxed{\text{Igloo}}$ function (for the function f from fig. B1) has therefore the form:

```
Igloo_f (t, newx)  =  f (FIRST newx, i)  WVR  (t = index)
                      FBY  Igloo_f  (t+1, NEXT newx) ;
result = Igloo_f (0, x) ;
```

A few remarks need to be made:

(a) Because of the non–existence of function parameters, a separate Igloo function must presently be written for each occurrence of currenting.

(b) Currenting automatically applies to the $\boxed{\text{WHERE}}$ expression a $\boxed{\text{WVR}}$ of the kind:

```
expr  WVR  (t = index)  WHERE  t IS CURRENT index ;  END
```

Recall that for any constant expression c:

```
(c WVR d) = c   # if d ever becomes TRUE.
```

The $\boxed{\text{WVR}}$ can therefore be omitted in the $\boxed{\text{Igloo}}$ function in any instance where the expression *expr* carries an $\boxed{\text{ASA}}$ on the outermost level.

As an example, take the function (from a famous Lucid prime program):

```
checkprime( n )   =   (  n < p*p   ASA   condition( p, n )  )
                      WHERE
                        n  IS CURRENT  n  ;
                      END  ;
```

According to the described method, this translates into:

```
chepri( k )       =   (  k < p*p   ASA   condition( p, k )  ) ;

checkprime( h )   =   chepri( FIRST h )  FBY  checkprime( NEXT h ) ;
                      # this is the simplified Igloo.
```

This can be simplified into:

```
checkprime( n )   =   (k < p*p   ASA   condition( p, k )  )
                      WHERE
                         k = FIRST n;
                      END  FBY checkprime( NEXT n );
```

End of example.

(c) If a $\boxed{\text{WHERE}}$ expression (here: f) has more than one currented variable, there is no need to nest $\boxed{\text{Igloo}}$ functions. Instead all these variables can be currented together. For example:

```
result  =  g (x, y)  WHERE  x IS CURRENT x;
                            y IS CURRENT y;  END ;
```

can be replaced by:

```
Igloo2 (t, newx, newy)  =  g (FIRST newx, FIRST newy)  WVR  (t = index)
                           FBY  Igloo2 (t+1, NEXT newx, NEXT newy) ;
result = Igloo2 (0, x, y) ;
```

Example (translation of currenting into the $\boxed{\text{Igloo}}$ form)

The following example is presented on page 28 of [AsW80]:

```
mom
  WHERE
    Avg (v)   = (s / (index+1))
                WHERE  s = v + (0 FBY s) ;  END ;

    m         = Avg (x),

    mom       = Avg ( (x-m) * (x-m) )
                WHERE  m IS CURRENT m ;  END ;
  END
```

The $\boxed{\text{mom}}$ in this example is the ***running moment*** (around the running average) of a given history x (there are more efficient ways of computing this). Using the $\boxed{\text{Igloo}}$ function, the example can be re-formulated, so that it contains no more "IS CURRENT":

```
mom
  WHERE
    Avg(v)         = (s / (index+1))
                     WHERE  s = v + (0 FBY s) ;  END ;
    m              = Avg (x);

    Body (m)       = Avg ( (x-m) * (x-m) );

    mom            = Igloo (0, m);
                     # the outside m is now currented

    Igloo (t, k) = ( Body (FIRST k)   WVR   (index = t) )
                     FBY
                     Igloo (t+1, NEXT k) ;
  END
```

B.4 Efficiency

Some people argue that the simulation of iteration by recursion leads to very inefficient code (i.e. many unnecessary computation steps will be carried out). However, as has been said before, such a claim can be invalidated by a good optimising compiler. The Igloo function is indeed easily optimised by applying some of the rules from chapter VI.

Because of the FBY , each new invocation of Igloo serves for the computation of *one* result daton. From a certain index on, all the results of the invocation will be determined by its *inner* re-invocation of Igloo with slightly changed parameters. Once the computation has progressed to the recursive re-invocation of Igloo (right operand of FBY), the whole left operand of FBY is superseded (i.e. not needed any longer). The actual parameters in the recursive call are simple modifications of the formal parameters: the storage cell for the constant t is simply incremented by one, and the index for the history k is advanced once (such operations can be *accumulated*).

Taken together, Igloo can be implemented by tail recursion Lucid-style (↑ 6.6). During the computation of any result daton (left operand of FBY) the index of history k is held constant, it is not affected by the computation inside f. Only a single-outport COPY node is therefore required as buffer for k (the buffer prevents the repeated evaluation of the same daton). As an example, the translation of fig. B1 (an arbitrary function whose operand 0 is currented whereas operand 1 is *not* currented) yields the following LUX code:

```
ACT Act_Igloo_Func ;
    LABEL 1 ;
    VAR
      superior, func, p0, p1, pp1, pii, pio : ACTOR   ;
      request   : MSGTYPE ;         index, i : INTEGER ;
      created   : BOOLEAN ;         result   : ANYTYPE ;
    BEGIN
      created := FALSE ;
      ( , , p0, p1) := RECEIVE FROM (Creator) ;

      pii := CREATE (Act_COPY_, 1) ;
      ( , pio)  :=  RECEIVE (pii)  ;
      SEND (DATON, p1)  TO  (pii)  ;

    REPEAT
      WHILE TRUE DO
      BEGIN                                                       :1
        (superior, request, index) := RECEIVE () ;                :1

        value0  := GetDaton (index, p0)       ;                   :1
        created := TRUE                       ;
        func    := CREATE (Act_Func)          ;
        SEND (DATON, value0, pio)   TO (func) ;

        FOR  i   := 1 TO index                                    :1
        DO   EXCEPTION (ADVANCE, i) TO (func) ;
                                                                  :1
        result  :=     GetDaton (index, func) ;
        EXCEPTION    (ADVANCE, finalindex) TO (func) ;
        created := FALSE                      ;                   :1
        SEND (DATON, result) TO (superior)    ;
      END ;

1:    (request, index) := Reveal ;
      EXCEPTION       (request,      index) TO (p0   ) ;
      IF   created
      THEN EXCEPTION (ADVANCE, finalindex) TO (func) ;
           created := FALSE ;
      RESET ;
    UNTIL   index = finalindex

    EXCEPTION        (request,      index) TO (pio ) ;
    END ;            (* End of Act_Igloo_Func .           *)
```

In LUX, it is even permitted to pass the act for f as a *parameter* to Act_Igloo_Func (like

$\boxed{\text{procedure}}$ or $\boxed{\text{function}}$ parameters in PASCAL); this relieves us from having a separate $\boxed{\text{Igloo}}$ for every instance of currenting. The function f itself is translated into:

```
ACT Act_Func ;
    LABEL 1 ;
    VAR
      superior, p1, pp1      : ACTOR    ;    request : MSGTYPE ;
      value0, value1, result : ANYTYPE  ;    index   : INTEGER ;
    BEGIN
      ( , , value0, p1) := RECEIVE FROM (Creator) ;
      pp1 := SEND (AUGMENT)  TO  (p1)  ;    (* † 6.3 *)

    REPEAT
      WHILE TRUE DO
      BEGIN                                                     :1
        (superior, request, index) := RECEIVE () ;              :1

        value1 := GetDaton (index, pp1)     ;                   :1
        result := ... value0 ... value1 ... ;                   :1
        SEND (DATON, result) TO (superior)  ;
      END ;             (* End of inner eternal loop.       *)

1:    (request, index) := Reveal ;
      EXCEPTION (request, index) TO (pp1) ;
      RESET ;
    UNTIL FALSE ;       (* End of outer eternal loop.       *)
    END ;               (* End of Act_Func .                *)
```

```
+----------------+
| Appendix C - 1 |
+----------------+
```

Complete listing of the program which translates any net or subnet from graph Lucid into LUX (for further detail see section 4.3.4). The program "Sieve" has been chosen for illustration.

```
program SieveTranslation (output) ;

const
     UDFops  = 30 ;
type
     oprange = 1..UDFops ;
  (* alfa    = packed array [1..10] of char ; *)
     NODEP   = ^ NODE ;                    (* node pointer *)
     NODE    = record
        ntype     : (otcopy, otcopytranslated, otinport, otother) ;
        nlabel    : integer ;
        ntext     : alfa    ;
        nnoofrefs : integer ;   (* number of node references (COPY!) *)
        nnoofops  : 0..UDFops ; (* number of node operands *)
        nop       : array [oprange] of NODEP ;
        ninitop   : array [oprange] of integer ;
      end ;


function NextLabel (var nodenumber : integer) : integer ;
   begin NextLabel  := nodenumber     ;              (* pseudo function *)
         nodenumber := nodenumber + 1 ;
   end ;


   function Translate (nuc : NODEP; var nodenumber : integer ) : integer ;
      forward ;


   procedure ScanOperands (nuc : NODEP; var nodenumber : integer) ;
      var  i     : integer ;
           nucop : NODEP   ;
      begin with nuc^ do
        for i := 1 to nnoofops
        do  begin
            nucop := nop[i] ;
            if   nucop^.ntype = otinport
            then begin  ninitop[i] := -nucop^.nlabel ;
                        dispose (nucop) ;
                 end
            else ninitop[i] := Translate(nucop, nodenumber) ;
      end   end ;      (* End of procedure 'ScanOperands'. *)
```

```
procedure NodeInitialisation (nuc : NODEP) ;
   var  i : integer ;
   begin with nuc^ do begin
     write ('   SEND (DATON, ') ;

     for i := 1 to nnoofops
     do  begin
         write ('node[',  ninitop[i]:2) ;
         if   i < nnoofops then  write ('], ') ;
         end ;

     writeln ('])  TO (node[', nlabel:2, ']) ;              (* ',
                    ntext, '*)');
   end end ; (* End of procedure 'NodeInitialisation'. *)


  function Translate ;        (* pseudo function *)
     (* The result of function 'Translate' is the subscript (label) of the
        node which will deliver the operand.  Note the split node labelling
        in the case of COPY nodes. *)

     var
        transl    : integer ;         (* new node will be node[(transl)] *)

     begin with nuc^ do begin

        transl    := NextLabel(nodenumber) ;
        Translate := transl ;         (* the function result! *)

             (* avoiding repeated COPY translation: *)
        if   ntype <> otcopytranslated
        then begin
             if   ntype = otcopy
             then begin  ntype  := otcopytranslated ;
                         nlabel := NextLabel(nodenumber) ;
                  end
             else        nlabel := transl ;

             writeln ('   node[', nlabel:2,
                      '] := CREATE(Act_', ntext, ') ;') ;

             ScanOperands (nuc, nodenumber) ;
             end;

        if   ntype = otcopytranslated
        then writeln ('   ( , , node[', transl:2,
                      ']) := RECEIVE FROM (node[', nlabel:2, ']) ;') ;
=== continued ===
```

=== continued ===

```
                nnoofrefs := nnoofrefs - 1 ;
           if   nnoofrefs = 0
           then begin
                if   nnoofops > 0  then  NodeInitialisation (nuc);
                dispose (nuc) ;

      end end end ;          (* End of function 'translate'. *)


procedure SegmentTranslate (nuc     : NODEP  ;
                            name    : alfa   ;
                            nodes   : integer;
                            inports : integer) ;
   var
     nodenumber : integer ;
     i          : integer ;

   begin
        writeln ('ACT Act_', name, ' ;') ;

   if   inports > 0
   then writeln ('   LABEL 1 ;') ;

        writeln ('   VAR') ;
        writeln ('     node     : ARRAY [', -inports:0, '..',
                 nodes-1:0, '] OF ACTOR ;') ;

   if   inports <= 0
   then writeln ('   BEGIN')
   else begin
        writeln ('     request : MSGTYPE ; index, skip : INTEGER ;') ;
        writeln ('   BEGIN') ;
        writeln ('   skip := 0 ;') ;
        write   ('   ( , ') ;
        for  i:=1 to inports  do  write (', node[', -i:0, ']') ;
        writeln (') := RECEIVE FROM (Creator) ;') ;
        writeln ;
        writeln ('   WHILE  Reveal = ADVANCE') ;
        writeln ('   DO BEGIN') ;
        writeln ('        (request, index) := Reveal ;') ;
        writeln ('        IF   index = finalindex') ;
        write   ('        THEN EXCEPTION (request, index) TO (') ;
        for  i := 1 to inports
        do   begin write   ('node[', -i:0) ;
                   if i < inports then write   ('], ') ;
             end ;
```

=== continued ===

```
=== continued ===
        writeln ('])') ;
        writeln ('       ELSE skip := skip + 1 ;') ;
        writeln ('       RESET ;') ;
        writeln ('     END ;') ;
        writeln ;
        end ;

   nodenumber := 0 ;
   i          := Translate (nuc, nodenumber) ; (* always yields zero *)

   if   inports < 1
   then writeln ('   Set_Priority (node[0], top_priority) ;')
   else begin
        writeln ('                                              :1') ;
        writeln ('1: Pass_Through (node[0], skip) ;') ;
        end ;

        writeln ('   END ;') ;
        writeln ;
        writeln ;
   end ;        (* End of procedure 'SegmentTranslate'. *)


procedure NodeDecl (var gln : NODEP; ntx : alfa; nops : integer) ;
   begin
     new (gln) ;
     with gln^ do
     begin
          ntext     := ntx      ;
     if   nops < 0
     then begin
          ntype     := otcopy   ;
          nnoofops  := 1        ;
          nnoofrefs := -nops    ;
          end
     else begin
          ntype     := otother ;
          nnoofops  := nops     ;
          nnoofrefs := 1        ;
     end  end
   end ;              (* End of procedure 'NodeDecl'. *)
```

```
(* ================= here starts the application =================== *)

(* 'RootDefine' and 'SieveDefine' place the Lucid graph of the
   entire Sieve program in store, ready for translation. *)

function RootDefine (i : integer) : NODEP ;     (* a pseudo-function *)
   var
     u : array [1..7] of NODEP ;
   begin
     i := i ;
     NodeDecl (u[1], 'Const_, 1',  0) ;
     NodeDecl (u[2],  'Plus_'   ,  2) ;
     NodeDecl (u[3], 'Const_, 2',  0) ;
     NodeDecl (u[4],   'Fby_'   ,  2) ;
     NodeDecl (u[5],  'Copy_, 2', -2) ;
     NodeDecl (u[6], 'Sieve'    ,  1) ;
     NodeDecl (u[7], 'Write_'   ,  1) ; (* 'Write_, "console"', *)
     RootDefine   := u[7] ;             (* highest ranking node *)

     u[2]^.nop[1] := u[1] ;     u[2]^.nop[2] := u[5] ;
     u[4]^.nop[1] := u[3] ;     u[4]^.nop[2] := u[2] ;
     u[5]^.nop[1] := u[4] ;
     u[6]^.nop[1] := u[5] ;
     u[7]^.nop[1] := u[6] ;
   end ;       (* End of function 'RootDefine'. *)


function SieveDefine (i : integer) : NODEP ;
   var
     u : array [1..9] of NODEP ;
   begin
     i := i ;
     NodeDecl (u[1],  'Copy_, 4', -4) ;
     NodeDecl (u[2], 'First_'   ,  1) ;
     NodeDecl (u[3],   'Mod_'   ,  2) ;
     NodeDecl (u[4], 'Const_, 0',  0) ;
     NodeDecl (u[5],    'Ne_'   ,  2) ;
     NodeDecl (u[6],   'Wvr_'   ,  2) ;
     NodeDecl (u[7], 'Sieve'    ,  1) ;
     NodeDecl (u[8],   'Fby_'   ,  2) ;
     NodeDecl (u[9], '{inport} ',  0) ; u[9]^.ntype  := otinport ;
                                         u[9]^.nlabel := 1 ;
     SieveDefine  := u[8] ;     (* highest ranking node *)
```

=== continued ===

=== continued ===

```
     u[1]^.nop[1] := u[9] ;
     u[2]^.nop[1] := u[1] ;
     u[3]^.nop[1] := u[1] ;      u[3]^.nop[2] := u[2] ;
     u[5]^.nop[1] := u[3] ;      u[5]^.nop[2] := u[4] ;
     u[6]^.nop[1] := u[1] ;      u[6]^.nop[2] := u[5] ;
     u[7]^.nop[1] := u[6] ;
     u[8]^.nop[1] := u[1] ;      u[8]^.nop[2] := u[7] ;
   end ;        (* End of function 'SieveDefine'. *)


begin
   writeln ;
   writeln ('(* LUX code for sieve" example: *)') ;
   writeln ;
   writeln ;
   SegmentTranslate (SieveDefine(0), 'Sieve', 12, 1) ;
        (* the "number of nodes" is equal to the number of nodes
           in the Lucid graph segment, except inport nodes,
           including COPY nodes, plus all COPY references. *)
   SegmentTranslate ( RootDefine(0),  'Root_',  9, 0) ;
end     (* End of main program. *)
.
```

This program produces the following output:

```
(* LUX code for "Sieve" example: *)


ACT Act_Sieve ;
   LABEL 1 ;
   VAR
     node    : ARRAY [-1..11] OF ACTOR ;
     request : MSGTYPE ; index, skip : INTEGER ;
   BEGIN
   skip := 0 ;
   ( , , node[-1]) := RECEIVE FROM (Creator) ;

   WHILE     Reveal = ADVANCE
   DO BEGIN
       (request, index) := Reveal ;
       IF   index  = finalindex
       THEN EXCEPTION (request, index) TO (node[-1])
       ELSE skip  := skip + 1 ;
       RESET ;
     END ;
```

=== continued ===

```
=== continued ===
   node[ 0] := CREATE(Act_Fby_       ) ;
   node[ 2] := CREATE(Act_Copy_, 4   ) ;
   ( , , node[ 1]) := RECEIVE FROM (node[ 2]) ;
   node[ 3] := CREATE(Act_Sieve      ) ;
   node[ 4] := CREATE(Act_Wvr_       ) ;
   ( , , node[ 5]) := RECEIVE FROM (node[ 2]) ;
   node[ 6] := CREATE(Act_Ne_        ) ;
   node[ 7] := CREATE(Act_Mod_       ) ;
   ( , , node[ 8]) := RECEIVE FROM (node[ 2]) ;
   node[ 9] := CREATE(Act_First_     ) ;
   ( , , node[10]) := RECEIVE FROM (node[ 2]) ;
   SEND (DATON, node[-1]) TO (node[ 2]) ;               (* Copy_, 4 *)
   SEND (DATON, node[10]) TO (node[ 9]) ;               (* First_   *)
   SEND (DATON, node[ 8], node[ 9]) TO (node[ 7]) ;     (* Mod_     *)
   node[11] := CREATE(Act_Const_, 0 ) ;
   SEND (DATON, node[ 7], node[11]) TO (node[ 6]) ;     (* Ne_      *)
   SEND (DATON, node[ 5], node[ 6]) TO (node[ 4]) ;     (* Wvr_     *)
   SEND (DATON, node[ 4]) TO (node[ 3]) ;               (* Sieve    *)
   SEND (DATON, node[ 1], node[ 3]) TO (node[ 0]) ;     (* Fby_     *)
                                                   :1
1: Pass_Through (node[0], skip) ;
   END ;


ACT Act_Root_ ;
   VAR
     node    : ARRAY [0..8] OF ACTOR ;
   BEGIN
   node[ 0] := CREATE(Act_Write_     ) ;
   node[ 1] := CREATE(Act_Sieve      ) ;
   node[ 3] := CREATE(Act_Copy_, 2   ) ;
   node[ 4] := CREATE(Act_Fby_       ) ;
   node[ 5] := CREATE(Act_Const_, 2  ) ;
   node[ 6] := CREATE(Act_Plus_      ) ;
   node[ 7] := CREATE(Act_Const_, 1  ) ;
   ( , , node[ 8]) := RECEIVE FROM (node[ 3]) ;
   SEND (DATON, node[ 7], node[ 8]) TO (node[ 6]) ;     (* Plus_    *)
   SEND (DATON, node[ 5], node[ 6]) TO (node[ 4]) ;     (* Fby_     *)
   ( , , node[ 2]) := RECEIVE FROM (node[ 3]) ;
   SEND (DATON, node[ 4]) TO (node[ 3]) ;               (* Copy_, 2 *)
   SEND (DATON, node[ 2]) TO (node[ 1]) ;               (* Sieve    *)
   SEND (DATON, node[ 1]) TO (node[ 0]) ;               (* Write_   *)
   Set_Priority (node[0], top_priority) ;
   END ;
```

```
+-----------------+
| Appendix D - 1 |
+-----------------+
```

OCCAM implementation (untested) of some Lucid operators

First the declaration of some constants:

```
DEF
  OTHERWISE = TRUE,
  NULLIFY   = 0   ,
  COMPUTE   = 1   ,
  ADVANCE   = 2   :
```

The following "PROC accept" should really be declared where indicted in the "PROC boolor", but has been pulled out for easier printing:

```
PROC accept (VALUE i) =
    IF
      dtn[1-i]   -- inspect daton value
        PAR
          exc[i]    ! NULLIFY;  index
          rplg      ! TRUE
      OTHERWISE
        ALT
          excg      ? request; xindex
            exc[i] ! NULLIFY;  index
          flag[i]  &
            rplg   ! dtn[i]     :
```

The "PROC boolor" is the counterpart for a LUX ACT. Here are first a few comments explaining the parameters:

```
   -- CHAN excg,      g -> boolor: exceptions
   --      cmpg,      g -> boolor: COMPUTE requests
   --      rplg,      boolor -> g: replies (daton values)
   --      exc[0],    boolor -> p0: exceptions
   --      cmp[0],    boolor -> p0: COMPUTE requests
   --      rpl[0]:    p0 -> boolor: replies
   --                 dto for p1
```

```
PROC boolor (CHAN excg,  cmpg,  rplg,     -- concurrent OR
                  exc[], cmp[], rpl[]) =

   VAR flag [1], dtn[1] :
   PAR
     PAR k = [0 FOR 1]
       WHILE TRUE
         SEQ
           rpl[k] ? dtn[k]
           flag[k] := TRUE

     WHILE TRUE
       VAR request, index, xindex :
       SEQ
         ALT
           excg         ? request; xindex
             SKIP
           cmpg         ? index
             SEQ
               flag[0] := FALSE
               flag[1] := FALSE
               PAR
                 request := COMPUTE
                 cmp[0] ! index
                 cmp[1] ! index

           +--------------------------------+
           | PROC accept (VALUE i) = ... :  |
           +--------------------------------+
               ALT
                 flag[0]    &
                   accept(1)
                 flag[1]    &
                   accept(0)
                 excg       ? request; xindex
                   PAR j = [0 FOR 1]
                     exc[j] ! NULLIFY;  index

         IF  request = ADVANCE   -- exception handling
               PAR i = [0 FOR 1]
                 exc[i] ! request; xindex

 :     -- End of PROC boolor
```

```
PROC write (CHAN excp, cmpp, rplp) =

   -- CHAN excp,     write -> p: exceptions
   --      cmpp,     write -> p: COMPUTE requests
   --      rplp:     p -> write: replies
   -- a channel output is assumed as predefined.

   VAR index, result :
   SEQ
     index := 0

     WHILE TRUE
       SEQ
         cmpp   ! index
         rplp   ? result
         output ! result
         index := index + 1
         excp   ! ADVANCE; index

   :     -- End of PROC write


PROC constant (CHAN excg, cmpg, rplg, VALUE const) =

   -- CHAN excg,      g -> constant: exceptions
   --      cmpg,      g -> constant: COMPUTE requests
   --      rplg:      constant -> g: replies (daton values)
   -- DEF  const = 4711 :   the value of the constant

   WHILE TRUE
     VAR request, index, xindex :
     ALT
       excg   ? request; xindex
         SKIP
       cmpg   ? index
         rplg ! const

   :     -- End of PROC constant
```

```
-- Occam implementation of the Lucid program:
--     TRUE or FALSE
-- main program:

CHAN excg, cmpg, rplg, exc[1], cmp[1], rpl[1] :
PAR
  write    (excg,   cmpg,    rplg)
  boolor   (excg,   cmpg,    rplg,
            exc,    cmp,     rpl)
  constant (exc[0], cmp[0],  rpl[0], TRUE)
  constant (exc[1], cmp[1],  rpl[1], FALSE)

-- End of example
```

There is a trade-off between the reduced number of request types in the Occam implementation of Lucid, and the lower number of channels in the LUX one (Occam channels are rather restrictive). The pattern matching of the LUX exception RECEIVE is replaced in the Occam implementation by ALTernative inputting through separate channels for (1) COMPUTE requests and (2) for all other requests. The absence in Occam of a counterpart for LUX doors makes it necessary to place exception inputs all over the process. Furthermore, Occam output statements cannot serve as guards (indeed, the general provision of such a mechanism is not trivial); this dictates a rather different result delivery strategy (channel "rpl") in Occam than in LUX.

The optimal scheduling, giving higher priority to exceptions, is not implied in the "boolor" example, above; it has to be resolved by means beyond present Occam. Anyway, Occam has ultimately been designed for for execution on a multiprocessor (an array of `transputers`), and scheduling is of minor importance in such a setting.

The state transitions of a 2-outport COPY enactor:

nr state	nr state why	nr state why	nr state why			
0	1" ... C.. . G	2" ... A.. . G	3" ... K.. . G	4 C.. C.. . gG	5" C.. A.. . gG	6" C.. K.. . gG
"	7 A.. A.. . gG	8" A.. K.. . gG	9 K.. K.. . gG			
1 ... C.. .	0 ... . G	1' C.. ... . gG	2' A.. ... . gG	3' K.. ... . gG	4 C.. C.. . g	5' A.. C.. . g
"	6' K.. C.. . g	10 ... ... C Gi	11" ... C.. C i	12' A.. ... C gGi	13' K.. ... C gGi	14 C.. C.. C gi
"	15' A.. C.. C gi	16' K.. C.. C gi				
2 ... A.. . u	7 A.. A.. . g	8' K.. A.. . g	119 ... ..1 . O	127 A.. ..1 . gO	131 K.. ..1 . gO	
3 ... K.. . u	8 A.. K.. . g	9 K.. K.. . g	122 ... K.1 . O	130 A.. K.1 . gO	134 K.. K.1 . gO	
4 C.. C.. .	0 ... . gG	1" ... C.. . g	10 ... ... C gGi	11" ... C.. C gi	14 C.. C.. C i	
5 C.. A.. . u	2 ... A.. . g	135 ... ..1 C gOi	139 C.. ..1 C Oi			
6 C.. K.. . u	3 ... K.. . g	138 ... K.1 C gOi	142 C.. K.1 C Oi			
7 A.. A.. . u	30 ... ... A oOi					
8 A.. K.. . u	169 ... K.1 A oOi					
9 K.. K.. . u	303 K.1 K.1 K oOi					
10 ... ... C u	0 i	2" ... A.. . Gi	3" ... K.. . Gi	7 A.. A.. . gGi	8" A.. K.. . gGi	9 K.. K.. . gGi
"	12" ... A.. C G	13" ... K.. C G	17 A.. A.. C gG	18" A.. K.. C gG	19 K.. K.. C gG	
11 ... C.. C	10 ... ... C G	11' C.. ... C gG	12' A.. ... C gG	13' K.. ... C gG	14 C.. C.. C g	15' A.. C.. C g
"	16' K.. C.. C g	20 ... ... D Gp	21" ... C.. D p	22' A.. ... D gGp	23' K.. ... D gGp	24 C.. C.. D gp
"	25' A.. C.. D gp	26' K.. C.. D gp				
12 ... A.. C u	17 A.. A.. C g	18' K.. A.. C g	119 ... ..1 . Oi	127 A.. ..1 . gOi	131 K.. ..1 . gOi	
13 ... K.. C u	18 A.. K.. C g	19 K.. K.. C g	122 ... K.1 . Oi	130 A.. K.1 . gOi	134 K.. K.1 . gOi	
14 C.. C.. C	10 ... ... C gG	11" ... C.. C g	20 ... ... D gGp	21" ... C.. D gp	24 C.. C.. D p	
15 C.. A.. C u	12 ... A.. C g	135 ... ..1 C gO	139 C.. ..1 C O			
16 C.. K.. C u	13 ... K.. C g	138 ... K.1 C gO	142 C.. K.1 C O			
17 A.. A.. C u	284 ..1 ..1 . oOi					
18 A.. K.. C u	287 ..1 K.1 . oOi					
19 K.. K.. C u	292 K.1 K.1 . oOi					
20 ... ... D	21" ... C.. D G	22" ... A.. D G	23" ... K.. D G	24 C.. C.. D gG	25" C.. A.. D gG	26" C.. K.. D gG
"	27 A.. A.. D gG	28" A.. K.. D gG	29 K.. K.. D gG	194 .1. .1. A i	195" .1. C1. A Gi	197" .1. A1. A Gi
"	198" .1. K1. A Gi	199 C1. C1. A gGi	201" C1. A1. A gGi	202" C1. K1. A gGi	206 A1. A1. A gGi	207" A1. K1. A gGi
"	208 K1. K1. A gGi					
21 ... C.. D	20 ... ... D G	21' C.. ... D gG	22' A.. ... D gG	23' K.. ... D gG	24 C.. C.. D g	25' A.. C.. D g
"	26' K.. C.. D g	196 .1. D1. A Oi	200 C1. D1. A gOi	204' A1. D1. A gOi	205' K1. D1. A gOi	
22 ... A.. D u	27 A.. A.. D g	28' K.. A.. D g	99' .1. ... A Oi	102' A1. ... A gOi	103' K1. ... A gOi	
23 ... K.. D u	28 A.. K.. D g	29 K.. K.. D g	271 .1. K.1 A Oi	280 A1. K.1 A gOi	283 K1. K.1 A gOi	
24 C.. C.. D	20 ... ... D gG	21" ... C.. D g	203 D1. D1. A oOi			
25 C.. A.. D u	22 ... A.. D g	99' .1. ... A gOi	100' C1. ... A Oi			
26 C.. K.. D u	23 ... K.. D g	271 .1. K.1 A gOi	274 C1. K.1 A Oi			
27 A.. A.. D u	30 ... ... A oOi					
28 A.. K.. D u	169 ... K.1 A oOi					
29 K.. K.. D u	303 K.1 K.1 K oOi					
30 ... ... A u	0 ... . p	2" ... A.. . Gp	3" ... K.. . Gp	7 A.. A.. . gGp	8" A.. K.. . gGp	9 K.. K.. . gGp
"	32" ... A.. A G	33" ... K.. A G	36 A.. A.. A gG	37" A.. K.. A gG	38 K.. K.. A gG	
31 ... C.. A u	0 ... . Gp	1 ... C.. . p	2' A.. ... . gGp	3' K.. ... . gGp	5' A.. C.. . gp	6' K.. C.. . gp
"	30 ... ... A G	32' A.. ... A gG	33' K.. ... A gG	34' A.. C.. A g	35' K.. C.. A g	
32 ... A.. A u	2 ... A.. . p	7 A.. A.. . gp	8' K.. A.. . gp	36 A.. A.. A g	37' K.. A.. A g	119 ... ..1 . Op
"	127 A.. ..1 . gOp	131 K.. ..1 . gOp	167 ... ..1 A O	173 A.. ..1 A gO	176 K.. ..1 A gO	
33 ... K.. A u	3 ... K.. . p	8 A.. K.. . gp	9 K.. K.. . gp	37 A.. K.. A g	38 K.. K.. A g	122 ... K.1 . Op
"	130 A.. K.1 . gOp	134 K.. K.1 . gOp	169 ... K.1 A O	175 A.. K.1 A gO	178 K.. K.1 A gO	
34 C.. A.. A u	2 ... A.. . gp	5 C.. A.. . p	32 ... A.. A g	119 ... ..1 . gOp	123 C.. ..1 . Op	167 ... ..1 A gO
"	170 C.. ..1 A O					
35 C.. K.. A u	3 ... K.. . gp	6 C.. K.. . p	33 ... K.. A g	122 ... K.1 . gOp	126 C.. K.1 . Op	169 ... K.1 A gO
"	172 C.. K.1 A O					
36 A.. A.. A u	7 A.. A.. . p	284 ..1 ..1 . oOp	297 ..1 ..1 A oO			
37 A.. K.. A u	8 A.. K.. . p	287 ..1 K.1 . oOp	299 ..1 K.1 A oO			

nr	state	nr	state	why	nr	state	why	nr	state	why									
38	K.. K.. A u	9	K.. K.. .	p	292	K.1 K.1 .	oOp	302	K.1 K.1 A	oO									
39	... .1. .	40	... C1. .	G	42	... A1. .	G	43	... K1. .	G	44	C.. .1. .	g	45	C.. C1. .	gG	47	C.. A1. .	gG
-		48	C.. K1. .	gG	49	A.. .1. .	g	50	A.. C1. .	gG	52	A.. A1. .	gG	53	A.. K1. .	gG	54	K.. .1. .	g
		55	K.. C1. .	gG	57	K.. A1. .	gG	58	K.. K1. .	gG									
40	... C1. .	39	... .1. .	G	41	... D1. .	O	44	C.. .1. .	gG	45	C.. C1. .	g	46	C.. D1. .	gO	49	A.. .1. .	gG
-		50	A.. C1. .	g	51	A.. D1. .	gO	54	K.. .1. .	gG	55	K.. C1. .	g	56	K.. D1. .	gO			
41	... D1. .	39	... .1. .	G	44	C.. .1. .	gG	46	C.. D1. .	g	49	A.. .1. .	gG	51	A.. D1. .	g	54	K.. .1. .	gG
-		56	K.. D1. .	g															
42	... A1. . u	0	... .	O	2'	A..	gO	3'	K..	gO	52	A.. A1. .	g	57	K.. A1. .	g			
43	... K1. . u	53	A.. K1. .	g	56	K.. K1. .	g	122	... K.1 .	O	130	A.. K.1 .	gO	134	K.. K.1 .	gO			
44	C.. .1. .	39	... .1. .	g	40	... C1. .	gG	42	... A1. .	gG	43	... K1. .	gG	45	C.. C1. .	G	47	C.. A1. .	G
-		48	C.. K1. .	G	59	... .1. C	gi	60	... C1. C	gGi	62	... A1. C	gGi	63	... K1. C	gGi	64	C.. .1. C	i
		65	C.. C1. C	Gi	67	C.. A1. C	Gi	68	C.. K1. C	Gi									
45	C.. C1. .	39	... .1. .	gG	40	... C1. .	g	44	C.. .1. .	G	61	... D1. C	gOi	66	C.. D1. C	Oi			
46	C.. D1. .	39	... .1. .	gG	41	... D1. .	g	44	C.. .1. .	G	59	... .1. C	gGi	61	... D1. C	gi	64	C.. .1. C	Gi
-		66	C.. D1. C	i															
47	C.. A1. . u	10	 C	gOi	11'	C.. ... C	Oi	42	... A1. .	g									
48	C.. K1. . u	43	... K1. .	g	138	... K.1 C	gOi	142	C.. K.1 C	Oi									
49	A.. .1. . u	52	A.. A1. .	G	53	A.. K1. .	G	209'	..1 .1. .	o	221'	..1 A1. .	Go	225'	..1 K1. .	Go			
50	A.. C1. . u	49	A.. .1. .	G	209'	..1 .1. .	Go	213'	..1 C1. .	o									
51	A.. D1. . u	217'	..1 D1. .	o															
52	A.. A1. . u	119'	..1	oO															
53	A.. K1. . u	169	... K.1 A	oOi															
54	K.. .1. . u	57	K.. A1. .	G	58	K.. K1. .	G	212'	K.1 .1. .	o	224'	K.1 A1. .	Go	228'	K.1 K1. .	Go			
55	K.. C1. . u	54	K.. .1. .	G	212'	K.1 .1. .	Go	216'	K.1 C1. .	o									
56	K.. D1. . u	220'	K.1 D1. .	o															
57	K.. A1. . u	122'	K.1	oO															
58	K.. K1. . u	303	K.1 K.1 K	oOi															
59	... .1. C u	39	... .1. .	i	42	... A1. .	Gi	43	... K1. .	Gi	49	A.. .1. .	gi	52	A.. A1. .	gGi	53	A.. K1. .	gGi
-		54	K.. .1. .	gi	57	K.. A1. .	gGi	58	K.. K1. .	gGi	62	... A1. C	G	63	... K1. C	G	69	A.. .1. C	g
-		72	A.. A1. C	gG	73	A.. K1. C	gG	74	K.. .1. C	g	77	K.. A1. C	gG	78	K.. K1. C	gG			
60	... C1. C u	39	... .1. .	Gi	40	... C1. .	i	49	A.. .1. .	gGi	50	A.. C1. .	gi	54	K.. .1. .	gGi	55	K.. C1. .	gi
-		59	... .1. C	G	69	A.. .1. C	gG	70	A.. C1. C	g	74	K.. .1. C	gG	75	K.. C1. C	g			
61	... D1. C u	41	... D1. .	i	51	A.. D1. .	gi	56	K.. D1. .	gi	71	A.. D1. C	g	76	K.. D1. C	g			
62	... A1. C u	0	... .	Oi	2'	A..	gOi	3'	K..	gOi	72	A.. A1. C	g	77	K.. A1. C	g			
63	... K1. C u	73	A.. K1. C	g	78	K.. K1. C	g	122	... K.1 .	Oi	130	A.. K.1 .	gOi	134	K.. K.1 .	gOi			
64	C.. .1. C	59	... .1. C	g	60	... C1. C	gG	62	... A1. C	gG	63	... K1. C	gG	65	C.. C1. C	G	67	C.. A1. C	G
-		68	C.. K1. C	G	79	... .1. D	gp	80	... C1. D	gGp	82	... A1. D	gGp	83	... K1. D	gGp	84	C.. .1. D	p
-		85	C.. C1. D	Gp	87	C.. A1. D	Gp	88	C.. K1. D	Gp									
65	C.. C1. C	59	... .1. C	gG	60	... C1. C	g	61	... D1. C	gO	64	C.. .1. C	G	66	C.. D1. C	O	79	... .1. D	gGp
-		80	... C1. D	gp	81	... D1. D	gOp	84	C.. .1. D	Gp	85	C.. C1. D	p	86	C.. D1. D	Op			
66	C.. D1. C	59	... .1. C	gG	61	... D1. C	g	64	C.. .1. C	G	79	... .1. D	gGp	81	... D1. D	gp	84	C.. .1. D	Gp
-		86	C.. D1. D	p															
67	C.. A1. C u	10	 C	gO	11'	C.. ... C	O	62	... A1. C	g									
68	C.. K1. C u	63	... K1. C	g	138	... K.1 C	gO	142	C.. K.1 C	O									
69	A.. .1. C u	72	A.. A1. C	G	73	A.. K1. C	G	209'	..1 .1. .	oi	221'	..1 A1. .	Goi	225'	..1 K1. .	Goi			
70	A.. C1. C u	69	A.. .1. C	G	209'	..1 .1. .	Goi	213'	..1 C1. .	oi									
71	A.. D1. C u	217'	..1 D1. .	oi															
72	A.. A1. C u	119'	..1	oOi															
73	A.. K1. C u	287	..1 K.1 .	oOi															
74	K.. .1. C u	77	K.. A1. C	G	78	K.. K1. C	G	212'	K.1 .1. .	oi	224'	K.1 A1. .	Goi	228'	K.1 K1. .	Goi			
75	K.. C1. C u	74	K.. .1. C	G	212'	K.1 .1. .	Goi	216'	K.1 C1. .	oi									
76	K.. D1. C u	220'	K.1 D1. .	oi															
77	K.. A1. C u	122'	K.1	oOi															
78	K.. K1. C u	292	K.1 K.1 .	oOi															

nr state	nr state why	nr state why	nr state why			
79 ... .1. D	80 ... C1. D G	82 ... A1. D G	83 ... K1. D G	84 C.. .1. D g	85 C.. C1. D gG	87 C.. A1. D gG
"	88 C.. K1. D gG	89 A.. .1. D g	90 A.. C1. D gG	92 A.. A1. D gG	93 A.. K1. D gG	94 K.. .1. D g
"	95 K.. C1. D gG	97 K.. A1. D gG	98 K.. K1. D gG			
80 ... C1. D	79 ... .1. D G	81 ... D1. D O	84 C.. .1. D gG	85 C.. C1. D g	86 C.. D1. D gO	89 A.. .1. D gG
"	90 A.. C1. D g	91 A.. D1. D gO	94 K.. .1. D gG	95 K.. C1. D g	96 K.. D1. D gO	
81 ... D1. D	79 ... .1. D G	84 C.. .1. D gG	86 C.. D1. D g	89 A.. .1. D gG	91 A.. D1. D g	94 K.. .1. D gG
"	96 K.. D1. D g					
82 ... A1. D u	20 ... ... D O	22' A.. ... D gO	23' K.. ... D gO	92 A.. A1. D g	97 K.. A1. D g	
83 ... K1. D u	93 A.. K1. D g	98 K.. K1. D g	154 ... K.1 D O	162 A.. K.1 D gO	166 K.. K.1 D gO	
84 C.. .1. D	79 ... .1. D g	80 ... C1. D gG	82 ... A1. D gG	83 ... K1. D gG	85 C.. C1. D G	87 C.. A1. D G
"	88 C.. K1. D G					
85 C.. C1. D	79 ... .1. D gG	80 ... C1. D g	81 ... D1. D gO	84 C.. .1. D G	86 C.. D1. D O	
86 C.. D1. D	79 ... .1. D gG	81 ... D1. D g	84 C.. .1. D G			
87 C.. A1. D u	20 ... ... D gO	21' C.. ... D O	82 ... A1. D g			
88 C.. K1. D u	83 ... K1. D g	154 ... K.1 D gO	158 C.. K.1 D O			
89 A.. .1. D u	92 A.. A1. D G	93 A.. K1. D G	249' ..1 .1. D o	261' ..1 A1. D Go	265' ..1 K1. D Go	
90 A.. C1. D u	89 A.. .1. D G	249' ..1 .1. D Go	253' ..1 C1. D o			
91 A.. D1. D u	257' ..1 D1. D o					
92 A.. A1. D u	151' ..1 ... D oO					
93 A.. K1. D u	293 ..1 K.1 D oO					
94 K.. .1. D u	97 K.. A1. D G	98 K.. K1. D G	252' K.1 .1. D o	264' K.1 A1. D Go	268' K.1 K1. D Go	
95 K.. C1. D u	94 K.. .1. D G	252' K.1 .1. D Go	256' K.1 C1. D o			
96 K.. D1. D u	260' K.1 D1. D o					
97 K.. A1. D u	154' K.1 ... D oO					
98 K.. K1. D u	303 K.1 K.1 K oOi					
99 ... .1. A u	39 ... .1. . p	42 ... A1. . Gp	43 ... K1. . Gp	49 A.. .1. . gp	52 A.. A1. . gGp	53 A.. K1. . gGp
"	54 K.. .1. . gp	57 K.. A1. . gGp	58 K.. K1. . gGp	102 ... A1. A G	103 ... K1. A G	109 A.. .1. A g
"	112 A.. A1. A gG	113 A.. K1. A gG	114 K.. .1. A g	117 K.. A1. A gG	118 K.. K1. A gG	
100 ... C1. A u	39 ... .1. . Gp	40 ... C1. . p	49 A.. .1. . gGp	50 A.. C1. . gp	54 K.. .1. . gGp	55 K.. C1. . gp
"	99 ... .1. A G	109 A.. .1. A gG	110 A.. C1. A g	114 K.. .1. A gG	115 K.. C1. A g	
101 ... D1. A u	41 ... D1. . p	51 A.. D1. . gp	56 K.. D1. . gp	111 A.. D1. A g	116 K.. D1. A g	
102 ... A1. A u	0 ... . Op	2' A.. ... . gOp	3' K.. ... . gOp	30 ... ... A O	32' A.. ... A gO	33' K.. ... A gO
"	42 ... A1. . p	52 A.. A1. . gp	57 K.. A1. . gp	112 A.. A1. A g	117 K.. A1. A g	
103 ... K1. A u	43 ... K1. . p	53 A.. K1. . gp	58 K.. K1. . gp	113 A.. K1. A g	118 K.. K1. A g	122 ... K.1 . Op
"	130 A.. K.1 . gOp	134 K.. K.1 . gOp	169 ... K.1 A O	175 A.. K.1 A gO	178 K.. K.1 A gO	
104 C.. .1. A u	39 ... .1. . gp	42 ... A1. . gGp	43 ... K1. . gGp	44 C.. .1. . p	47 C.. A1. . Gp	48 C.. K1. . Gp
"	99 ... .1. A g	102 ... A1. A gG	103 ... K1. A gG	107 C.. A1. A G	108 C.. K1. A G	
105 C.. C1. A u	39 ... .1. . gGp	40 ... C1. . gp	44 C.. .1. . Gp	45 C.. C1. . p	99 ... .1. A gG	100 ... C1. A g
"	104 C.. .1. A G					
106 C.. D1. A u	41 ... D1. . gp	46 C.. D1. . p	101 ... D1. A g			
107 C.. A1. A u	0 ... . gOp	1' C.. ... . Op	30 ... ... A gO	31' C.. ... A O	42 ... A1. . gp	47 C.. A1. . p
"	102 ... A1. A g					
108 C.. K1. A u	43 ... K1. . gp	48 C.. K1. . p	103 ... K1. A g	122 ... K.1 . gOp	126 C.. K.1 . Op	169 ... K.1 A gO
"	172 C.. K.1 A O					
109 A.. .1. A u	49 A.. .1. . p	52 A.. A1. . Gp	53 A.. K1. . Gp	112 A.. A1. A G	113 A.. K1. A G	209' ..1 .1. . op
"	221' ..1 A1. . Gop	225' ..1 K1. . Gop	269' ..1 .1. A o	278' ..1 A1. A Go	281' ..1 K1. A Go	
110 A.. C1. A u	49 A.. .1. . Gp	50 A.. C1. . p	109 A.. .1. A G	209' ..1 .1. . Gop	213' ..1 C1. . op	269' ..1 .1. A Go
"	272' ..1 C1. A o					
111 A.. D1. A u	51 A.. D1. . p	217' ..1 D1. . op	275' ..1 D1. A o			
112 A.. A1. A u	52 A.. A1. . p	119' ..1 ... . oOp	167' ..1 ... A oO			
113 A.. K1. A u	53 A.. K1. . p	287 ..1 K.1 . oOp	299 ..1 K.1 A oO			
114 K.. .1. A u	54 K.. .1. . p	57 K.. A1. . Gp	58 K.. K1. . Gp	117 K.. A1. A G	118 K.. K1. A G	212' K.1 .1. . op
"	224' K.1 A1. . Gop	228' K.1 K1. . Gop	271' K.1 .1. A o	280' K.1 A1. A Go	283' K.1 K1. A Go	
115 K.. C1. A u	54 K.. .1. . Gp	55 K.. C1. . p	114 K.. .1. A G	212' K.1 .1. . Gop	216' K.1 C1. . op	271' K.1 .1. A Go
"	274' K.1 C1. A o					
116 K.. D1. A u	56 K.. D1. . p	220' K.1 D1. . op	277' K.1 D1. A o			

nr	state	nr	state	why	nr	state	why	nr	state	why									
117	K.. A1. A u	57	K.. A1. .	p	122'	K.1	oOp	169'	K.1 ... A	oO									
118	K.. K1. A u	58	K.. K1. .	p	292	K.1 K.1 .	oOp	302	K.1 K.1 A	oO									
119	... ..1 .	120	... C.1 .	G	121	... A.1 .	G	122	... K.1 .	G	123	C.. ..1 .	g	124	C.. C.1 .	gG	125	C.. A.1 .	gG
-		126	C.. K.1 .	gG	127	A.. ..1 .	g	128	A.. C.1 .	gG	129	A.. A.1 .	gG	130	A.. K.1 .	gG	131	K.. ..1 .	g
-		132	K.. C.1 .	gG	133	K.. A.1 .	gG	134	K.. K.1 .	gG									
120	... C.1 .	119	... ..1 .	G	123	C.. ..1 .	gG	124	C.. C.1 .	g	127	A.. ..1 .	gG	128	A.. C.1 .	g	131	K.. ..1 .	gG
-		132	K.. C.1 .	g	135	... ..1 C	Gi	136	... C.1 C	i	139	C.. ..1 C	gGi	140	C.. C.1 C	gi	143	A.. ..1 C	gGi
-		144	A.. C.1 C	gi	147	K.. ..1 C	gGi	148	K.. C.1 C	gi									
121	... A.1 . u	119-	... ..2 .	O	127-	A.. ..2 .	gO	129	A.. A.1 .	g	131-	K.. ..2 .	gO	133	K.. A.1 .	g			
122	... K.1 .	126	C.. K.1 .	g	130	A.. K.1 .	g	134	K.. K.1 .	g									
123	C.. ..1 .	119	... ..1 .	g	120	... C.1 .	gG	121	... A.1 .	gG	122	... K.1 .	gG	124	C.. C.1 .	G	125	C.. A.1 .	G
-		126	C.. K.1 .	G	135	... ..1 C	gi	136	... C.1 C	gGi	137	... A.1 C	gGi	138	... K.1 C	gGi	139	C.. ..1 C	i
-		140	C.. C.1 C	Gi	141	C.. A.1 C	Gi	142	C.. K.1 C	Gi									
124	C.. C.1 .	119	... ..1 .	gG	120	... C.1 .	g	123	C.. ..1 .	G	135	... ..1 C	gGi	136	... C.1 C	gi	139	C.. ..1 C	Gi
-		140	C.. C.1 C	i															
125	C.. A.1 . u	121	... A.1 .	g	135-	... ..2 C	gOi	139-	C.. ..2 C	Oi									
126	C.. K.1 .	122	... K.1 .	g	138	... K.1 C	gi	142	C.. K.1 C	i									
127	A.. ..1 . u	30	... ... A	oi	32	... A.. A	Goi	33	... K.. A	Goi	129	A.. A.1 .	G	130	A.. K.1 .	G			
128	A.. C.1 . u	30	... ... A	Goi	31	... C.. A	oi	127	A.. ..1 .	G									
129	A.. A.1 . u	167	... ..1 A	oOi															
130	A.. K.1 . u	169	... K.1 A	oi															
131	K.. ..1 . u	133	K.. A.1 .	G	134	K.. K.1 .	G	169'	K.1 ... A	oi	175'	K.1 A.. A	Goi	178'	K.1 K.. A	Goi			
132	K.. C.1 . u	131	K.. ..1 .	G	169'	K.1 ... A	Goi	172'	K.1 C.. A	oi									
133	K.. A.1 . u	299'	K.1 ..1 A	oOi															
134	K.. K.1 . u	303	K.1 K.1 K	oi															
135	... ..1 C u	119	... ..1 .	i	121	... A.1 .	Gi	122	... K.1 .	Gi	127	A.. ..1 .	gi	129	A.. A.1 .	gGi	130	A.. K.1 .	gGi
-		131	K.. ..1 .	gi	133	K.. A.1 .	gGi	134	K.. K.1 .	gGi	137	... A.1 C	G	138	... K.1 C	G	143	A.. ..1 C	g
-		145	A.. A.1 C	gG	146	A.. K.1 C	gG	147	K.. ..1 C	g	149	K.. A.1 C	gG	150	K.. K.1 C	gG			
136	... C.1 C	135	... ..1 C	G	139	C.. ..1 C	gG	140	C.. C.1 C	g	143	A.. ..1 C	gG	144	A.. C.1 C	g	147	K.. ..1 C	gG
-		148	K.. C.1 C	g	151	... ..1 D	Gp	152	... C.1 D	p	155	C.. ..1 D	gGp	156	C.. C.1 D	gp	159	A.. ..1 D	gGp
-		160	A.. C.1 D	gp	163	K.. ..1 D	gGp	164	K.. C.1 D	gp									
137	... A.1 C u	119-	... ..2 .	Oi	127-	A.. ..2 .	gOi	131-	K.. ..2 .	gOi	145	A.. A.1 C	g	149	K.. A.1 C	g			
138	... K.1 C u	122	... K.1 .	i	130	A.. K.1 .	gi	134	K.. K.1 .	gi	146	A.. K.1 C	g	150	K.. K.1 C	g			
139	C.. ..1 C	135	... ..1 C	g	136	... C.1 C	gG	137	... A.1 C	gG	138	... K.1 C	gG	140	C.. C.1 C	G	141	C.. A.1 C	G
-		142	C.. K.1 C	G	151	... ..1 D	gp	152	... C.1 D	gGp	153	... A.1 D	gGp	154	... K.1 D	gGp	155	C.. ..1 D	p
-		156	C.. C.1 D	Gp	157	C.. A.1 D	Gp	158	C.. K.1 D	Gp									
140	C.. C.1 C	135	... ..1 C	gG	136	... C.1 C	g	139	C.. ..1 C	G	151	... ..1 D	gGp	152	... C.1 D	gp	155	C.. ..1 D	Gp
-		156	C.. C.1 D	p															
141	C.. A.1 C u	135-	... ..2 C	gO	137	... A.1 C	g	139-	C.. ..2 C	O									
142	C.. K.1 C	138	... K.1 C	g	154	... K.1 D	gp	158	C.. K.1 D	p									
143	A.. ..1 C u	145	A.. A.1 C	G	146	A.. K.1 C	G	284	..1 ..1 .	oi	286	..1 A.1 .	Goi	287	..1 K.1 .	Goi			
144	A.. C.1 C u	143	A.. ..1 C	G	284	..1 ..1 .	Goi	285	..1 C.1 .	oi									
145	A.. A.1 C u	284-	..1 ..2 .	oOi															
146	A.. K.1 C u	287	..1 K.1 .	oi															
147	K.. ..1 C u	149	K.. A.1 C	G	150	K.. K.1 C	G	287'	K.1 ..1 .	oi	291'	K.1 A.1 .	Goi	292	K.1 K.1 .	Goi			
148	K.. C.1 C u	147	K.. ..1 C	G	287'	K.1 ..1 .	Goi	289'	K.1 C.1 .	oi									
149	K.. A.1 C u	287-	K.1 ..2 .	oOi															
150	K.. K.1 C u	292	K.1 K.1 .	oi															
151	... ..1 D	99'	.1. ... A	i	100'	C1. ... A	gi	102'	A1. ... A	gi	103'	K1. ... A	gi	104'	.1. C.. A	Gi	105'	C1. C.. A	gGi
-		107'	A1. C.. A	gGi	108'	K1. C.. A	gGi	109'	.1. A.. A	Gi	110'	C1. A.. A	gGi	112'	A1. A.. A	gGi	113'	K1. A.. A	gGi
-		114'	.1. K.. A	Gi	115'	C1. K.. A	gGi	117'	A1. K.. A	gGi	118'	K1. K.. A	gGi	152	... C.1 D	G	153	... A.1 D	G
-		154	... K.1 D	G	155	C.. ..1 D	g	156	C.. C.1 D	gG	157	C.. A.1 D	gG	158	C.. K.1 D	gG	159	A.. ..1 D	g
-		160	A.. C.1 D	gG	161	A.. A.1 D	gG	162	A.. K.1 D	gG	163	K.. ..1 D	g	164	K.. C.1 D	gG	165	K.. A.1 D	gG
-		166	K.. K.1 D	gG															
152	... C.1 D	99'	.1. ... A	Gi	100'	C1. ... A	gGi	102'	A1. ... A	gGi	103'	K1. ... A	gGi	104'	.1. C.. A	i	105'	C1. C.. A	gi
-		107'	A1. C.. A	gi	108'	K1. C.. A	gi	151	... ..1 D	G	155	C.. ..1 D	gG	156	C.. C.1 D	g	159	A.. ..1 D	gG
-		160	A.. C.1 D	g	163	K.. ..1 D	gG	164	K.. C.1 D	g									

nr state	nr state why	nr state why	nr state why			
153 ... A.1 D u	161 A.. A.1 D g	165 K.. A.1 D g	269 .1. ..1 A Oi	278 A1. ..1 A gOi	281 K1. ..1 A gOi	
154 ... K.1 D	158 C.. K.1 D g	162 A.. K.1 D g	166 K.. K.1 D g	271 .1. K.1 A i	274 C1. K.1 A gi	280 A1. K.1 A gi
	283 K1. K.1 A gi					
155 C.. ..1 D	101' D1. ... A oi	106' D1. C.. A Goi	111' D1. A.. A Goi	116' D1. K.. A Goi	151 ... ..1 D g	152 ... C.1 D gG
	153 ... A.1 D gG	154 ... K.1 D gG	156 C.. C.1 D G	157 C.. A.1 D G	158 C.. K.1 D G	
156 C.. C.1 D	101' D1. ... A Goi	106' D1. C.. A oi	151 ... ..1 D gG	152 ... C.1 D g	155 C.. ..1 D G	
157 C.. A.1 D u	153 ... A.1 D g	269 .1. ..1 A gOi	272 C1. ..1 A Oi			
158 C.. K.1 D	154 ... K.1 D g	277 D1. K.1 A oi				
159 A.. ..1 D u	30 ... ... A oi	32 ... A.. A Goi	33 ... K.. A Goi	161 A.. A.1 D G	162 A.. K.1 D G	
160 A.. C.1 D u	30 ... ... A Goi	31 ... C.. A oi	159 A.. ..1 D G			
161 A.. A.1 D u	167 ... ..1 A oOi					
162 A.. K.1 D u	169 ... K.1 A oi					
163 K.. ..1 D u	165 K.. A.1 D G	166 K.. K.1 D G	169' K.1 ... A oi	175' K.1 A.. A Goi	178' K.1 K.. A Goi	
164 K.. C.1 D u	163 K.. ..1 D G	169' K.1 ... A Goi	172' K.1 C.. A oi			
165 K.. A.1 D u	299' K.1 ..1 A oOi					
166 K.. K.1 D u	303 K.1 K.1 K oi					
167 ... ..1 A u	119 ... ..1 . p	121 ... A.1 . Gp	122 ... K.1 . Gp	127 A.. ..1 . gp	129 A.. A.1 . gGp	130 A.. K.1 . gGp
	131 K.. ..1 . gp	133 K.. A.1 . gGp	134 K.. K.1 . gGp	168 ... A.1 A G	169 ... K.1 A G	173 A.. ..1 A g
	174 A.. A.1 A gG	175 A.. K.1 A gG	176 K.. ..1 A g	177 K.. A.1 A gG	178 K.. K.1 A gG	
168 ... A.1 A u	119- ... ..2 . Op	121 ... A.1 . p	127- A.. ..2 . gOp	129 A.. A.1 . gp	131- K.. ..2 . gOp	133 K.. A.1 . gp
	167- ... ..2 A O	173- A.. ..2 A gO	174 A.. A.1 A g	176- K.. ..2 A gO	177 K.. A.1 A g	
169 ... K.1 A u	122 ... K.1 . p	130 A.. K.1 . gp	134 K.. K.1 . gp	175 A.. K.1 A g	178 K.. K.1 A g	
170 C.. ..1 A u	119 ... ..1 . gp	121 ... A.1 . gGp	122 ... K.1 . gGp	123 C.. ..1 . p	125 C.. A.1 . Gp	126 C.. K.1 . Gp
	167 ... ..1 A g	168 ... A.1 A gG	169 ... K.1 A gG	171 C.. A.1 A G	172 C.. K.1 A G	
171 C.. A.1 A u	119- ... ..2 . gOp	121 ... A.1 . gp	123- C.. ..2 . Op	125 C.. A.1 . p	167- ... ..2 A gO	168 ... A.1 A g
	170- C.. ..2 A O					
172 C.. K.1 A u	122 ... K.1 . gp	126 C.. K.1 . p	169 ... K.1 A g			
173 A.. ..1 A u	127 A.. ..1 . p	129 A.. A.1 . Gp	130 A.. K.1 . Gp	174 A.. A.1 A G	175 A.. K.1 A G	284 ..1 ..1 . op
	286 ..1 A.1 . Gop	287 ..1 K.1 . Gop	297 ..1 ..1 A o	298 ..1 A.1 A Go	299 ..1 K.1 A Go	
174 A.. A.1 A u	129 A.. A.1 . p	284- ..1 ..2 . oOp	297- ..1 ..2 A oO			
175 A.. K.1 A u	130 A.. K.1 . p	287 ..1 K.1 . op	299 ..1 K.1 A o			
176 K.. ..1 A u	131 K.. ..1 . p	133 K.. A.1 . Gp	134 K.. K.1 . Gp	177 K.. A.1 A G	178 K.. K.1 A G	287' K.1 ..1 . op
	291' K.1 A.1 . Gop	292 K.1 K.1 . Gop	299' K.1 ..1 A o	301' K.1 A.1 A Go	302 K.1 K.1 A Go	
177 K.. A.1 A u	133 K.. A.1 . p	287- K.1 ..2 . oOp	299- K.1 ..2 A oO			
178 K.. K.1 A u	134 K.. K.1 . p	292 K.1 K.1 . op	302 K.1 K.1 A o			
179 .1. .1. .	180" .1. C1. . G	182" .1. A1. . G	183" .1. K1. . G	184 C1. C1. . gG	186" C1. A1. . gG	187" C1. K1. . gG
	191 A1. A1. . gG	192" A1. K1. . gG	193 K1. K1. . gG			
180 .1. C1. .	179 .1. .1. . G	180' C1. .1. . gG	181 .1. D1. . O	182' A1. .1. . gG	183' K1. .1. . gG	184 C1. C1. . g
	185 C1. D1. . gO	186' A1. C1. . g	187' K1. C1. . g	189' A1. D1. . gO	190' K1. D1. . gO	
181 .1. D1. .	179 .1. .1. . G	180' C1. .1. . gG	182' A1. .1. . gG	183' K1. .1. . gG	185 C1. D1. . g	189' A1. D1. . g
	190' K1. D1. . g					
182 .1. A1. . u	39' .1. ... . O	42' A1. ... . gO	43' K1. ... . gO	191 A1. A1. . g	192' K1. A1. . g	
183 .1. K1. . u	192 A1. K1. . g	193 K1. K1. . g	212 .1. K.1 . O	224 A1. K.1 . gO	228 K1. K.1 . gO	
184 C1. C1. .	179 .1. .1. . gG	180" .1. C1. . g	188 D1. D1. . oO			
185 C1. D1. .	179 .1. .1. . gG	180' C1. .1. . G	181" .1. D1. . g	188 D1. D1. . o		
186 C1. A1. . u	39' .1. ... . gO	40' C1. ... . O	182 .1. A1. . g			
187 C1. K1. . u	183 .1. K1. . g	212 .1. K.1 . gO	216 C1. K.1 . O			
188 D1. D1. .	179 .1. .1. . gG	181" .1. D1. . g				
189 D1. A1. . u	41' D1. ... . O					
190 D1. K1. . u	220 D1. K.1 . O					
191 A1. A1. . u	0 ... . oO					
192 A1. K1. . u	122 ... K.1 . oO					
193 K1. K1. . u	303 K.1 K.1 K oOi					
194 .1. .1. A u	179 .1. .1. . p	182" .1. A1. . Gp	183" .1. K1. . Gp	191 A1. A1. . gGp	192" A1. K1. . gGp	193 K1. K1. . gGp
	197" .1. A1. A G	198" .1. K1. A G	206 A1. A1. A gG	207" A1. K1. A gG	208 K1. K1. A gG	
195 .1. C1. A u	179 .1. .1. . Gp	180 .1. C1. . p	182' A1. .1. . gGp	183' K1. .1. . gGp	186' A1. C1. . gp	187' K1. C1. . gp
	194 .1. .1. A G	197' A1. .1. A gG	198' K1. .1. A gG	201' A1. C1. A g	202' K1. C1. A g	

nr	state	nr	state	why	nr	state	why	nr	state	why									
196	.1. D1. A u	181	.1. D1. .	p	189'	A1. D1. .	gp	190'	K1. D1. .	gp	204'	A1. D1. A	g	205'	K1. D1. A	g			
197	.1. A1. A u	39'	.1.	Op	42'	A1.	gOp	43'	K1.	gOp	99'	.1. ... A	O	102'	A1. ... A	gO	103'	K1. ... A	gO
		182	.1. A1. .	p	191	A1. A1. .	gp	192'	K1. A1. .	gp	206	A1. A1. A	g	207'	K1. A1. A	g			
198	.1. K1. A u	183	.1. K1. .	p	192	A1. K1. .	gp	193	K1. K1. .	gp	207	A1. K1. A	g	208	K1. K1. A	g	212	.1. K.1 .	Op
		224	A1. K.1 .	gOp	228	K1. K.1 .	gOp	271	.1. K.1 A	O	280	A1. K.1 A	gO	283	K1. K.1 A	gO			
199	C1. C1. A u	179	.1. .1. .	gGp	180"	.1. C1. .	gp	184	C1. C1. .	p	194	.1. .1. A	gG	195"	.1. C1. A	g			
200	C1. D1. A u	181	.1. D1. .	gp	185	C1. D1. .	p	196	.1. D1. A	g									
201	C1. A1. A u	39'	.1.	gOp	40'	C1.	Op	99'	.1. ... A	gO	100'	C1. ... A	O	182	.1. A1. .	gp	186	C1. A1. .	p
		197	.1. A1. A	g															
202	C1. K1. A u	183	.1. K1. .	gp	187	C1. K1. .	p	198	.1. K1. A	g	212	.1. K.1 .	gOp	216	C1. K.1 .	Op	271	.1. K.1 A	gO
		274	C1. K.1 A	O															
203	D1. D1. A u	188	D1. D1. .	p															
204	D1. A1. A u	41'	D1.	Op	101'	D1. ... A	O	189	D1. A1. .	p									
205	D1. K1. A u	130	D1. K1. .	p	220	D1. K.1 .	Op	277	D1. K.1 A	O									
206	A1. A1. A u	0	... .	oOp	39	... ... A	oO	191	A1. A1. .	p									
207	A1. K1. A u	122	... K.1 .	oOp	169	... K.1 A	oO	192	A1. K1. .	p									
208	K1. K1. A u	193	K1. K1. .	p	292	K.1 K.1 .	oOp	302	K.1 K.1 A	oO									
209	.1. ..1 .	210	.1. C.1 .	G	211	.1. A.1 .	G	212	.1. K.1 .	G	213	C1. ..1 .	g	214	C1. C.1 .	gG	215	C1. A.1 .	gG
		216	C1. K.1 .	gG	221	A1. ..1 .	g	222	A1. C.1 .	gG	223	A1. A.1 .	gG	224	A1. K.1 .	gG	225	K1. ..1 .	g
		226	K1. C.1 .	gG	227	K1. A.1 .	gG	228	K1. K.1 .	gG									
210	.1. C.1 .	209	.1. ..1 .	G	213	C1. ..1 .	gG	214	C1. C.1 .	g	221	A1. ..1 .	gG	222	A1. C.1 .	g	225	K1. ..1 .	gG
		226	K1. C.1 .	g	229	.1. ..1 C	Gi	230	.1. C.1 C	i	233	C1. ..1 C	gGi	234	C1. C.1 C	gi	241	A1. ..1 C	gGi
		242	A1. C.1 C	gi	245	K1. ..1 C	gGi	246	K1. C.1 C	gi									
211	.1. A.1 . u	209-	.1. ..2 .	O	221-	A1. ..2 .	gO	223	A1. A.1 .	g	225-	K1. ..2 .	gO	227	K1. A.1 .	g			
212	.1. K.1 .	216	C1. K.1 .	g	224	A1. K.1 .	g	228	K1. K.1 .	g									
213	C1. ..1 .	209	.1. ..1 .	g	210	.1. C.1 .	gG	211	.1. A.1 .	gG	212	.1. K.1 .	gG	214	C1. C.1 .	G	215	C1. A.1 .	G
		216	C1. K.1 .	G	217	D1. ..1 .	o	218	D1. C.1 .	Go	219	D1. A.1 .	Go	220	D1. K.1 .	Go			
214	C1. C.1 .	209	.1. ..1 .	gG	210	.1. C.1 .	g	213	C1. ..1 .	G	237	D1. ..1 C	Goi	238	D1. C.1 C	oi			
215	C1. A.1 . u	209-	.1. ..2 .	gO	211	.1. A.1 .	g	213-	C1. ..2 .	O									
216	C1. K.1 .	212	.1. K.1 .	g	220	D1. K.1 .	o												
217	D1. ..1 .	209	.1. ..1 .	g	210	.1. C.1 .	gG	211	.1. A.1 .	gG	212	.1. K.1 .	gG	218	D1. C.1 .	G	219	D1. A.1 .	G
		220	D1. K.1 .	G															
218	D1. C.1 .	209	.1. ..1 .	gG	210	.1. C.1 .	g	217	D1. ..1 .	G	229	.1. ..1 C	gGi	230	.1. C.1 C	gi	237	D1. ..1 C	Gi
		238	D1. C.1 C	i															
219	D1. A.1 . u	217-	D1. ..2 .	O															
220	D1. K.1 .	212	.1. K.1 .	g															
221	A1. ..1 . u	119	... ..1 .	o	121	... A.1 .	Go	122	... K.1 .	Go	223	A1. A.1 .	G	224	A1. K.1 .	G			
222	A1. C.1 . u	135	... ..1 C	Goi	136	... C.1 C	oi	221	A1. ..1 .	G									
223	A1. A.1 . u	119-	... ..2 .	oO															
224	A1. K.1 . u	122	... K.1 .	o															
225	K1. ..1 . u	169'	K.1 ... A	oi	175'	K.1 A.. A	Goi	178'	K.1 K.. A	Goi	227	K1. A.1 .	G	228	K1. K.1 .	G			
226	K1. C.1 . u	169'	K.1 ... A	Goi	172'	K.1 C.. A	oi	225	K1. ..1 .	G									
227	K1. A.1 . u	299'	K.1 ..1 A	oOi															
228	K1. K.1 . u	303	K.1 K.1 K	oi															
229	.1. ..1 C u	209	.1. ..1 .	i	211	.1. A.1 .	Gi	212	.1. K.1 .	Gi	221	A1. ..1 .	gi	223	A1. A.1 .	gGi	224	A1. K.1 .	gGi
		225	K1. ..1 .	gi	227	K1. A.1 .	gGi	228	K1. K.1 .	gGi	231	.1. A.1 C	G	232	.1. K.1 C	G	241	A1. ..1 C	g
		243	A1. A.1 C	gG	244	A1. K.1 C	gG	245	K1. ..1 C	g	247	K1. A.1 C	gG	248	K1. K.1 C	gG			
230	.1. C.1 C	229	.1. ..1 C	G	233	C1. ..1 C	gG	234	C1. C.1 C	g	241	A1. ..1 C	gG	242	A1. C.1 C	g	245	K1. ..1 C	gG
		246	K1. C.1 C	g	249	.1. ..1 D	Gp	250	.1. C.1 D	p	253	C1. ..1 D	gGp	254	C1. C.1 D	gp	261	A1. ..1 D	gGp
		262	A1. C.1 D	gp	265	K1. ..1 D	gGp	266	K1. C.1 D	gp									
231	.1. A.1 C u	209-	.1. ..2 .	Oi	221-	A1. ..2 .	gOi	225-	K1. ..2 .	gOi	243	A1. A.1 C	g	247	K1. A.1 C	g			
232	.1. K.1 C u	212	.1. K.1 .	i	224	A1. K.1 .	gi	228	K1. K.1 .	gi	244	A1. K.1 C	g	248	K1. K.1 C	g			
233	C1. ..1 C u	209	.1. ..1 .	gi	211	.1. A.1 .	gGi	212	.1. K.1 .	gGi	213	C1. ..1 .	i	215	C1. A.1 .	Gi	216	C1. K.1 .	Gi
		229	.1. ..1 C	g	231	.1. A.1 C	gG	232	.1. K.1 C	gG	235	C1. A.1 C	G	236	C1. K.1 C	G			
234	C1. C.1 C	229	.1. ..1 C	gG	230	.1. C.1 C	g	233	C1. ..1 C	G	237	D1. ..1 C	Go	238	D1. C.1 C	o	249	.1. ..1 D	gGp
		250	.1. C.1 D	gp	253	C1. ..1 D	Gp	254	C1. C.1 D	p	257	D1. ..1 D	Gop	258	D1. C.1 D	op			
235	C1. A.1 C u	209-	.1. ..2 .	gOi	213-	C1. ..2 .	Oi	231	.1. A.1 C	g									

nr	state	nr	state	why	nr	state	why	nr	state	why						
236	C1. K.1 C u	212	.1. K.1 .	gi	216	C1. K.1 .	i	232	.1. K.1 C	g						
237	D1. ..1 C u	217	D1. ..1 .	i	219	D1. A.1 .	Gi	220	D1. K.1 .	Gi	239	D1. A.1 C G	240	D1. K.1 C G		
238	D1. C.1 C	229	.1. ..1 C	gG	230	.1. C.1 C	g	237	D1. ..1 C	G	249	.1. ..1 D gGp	250	.1. C.1 D gp	257	D1. ..1 D Gp
		258	D1. C.1 D	p												
239	D1. A.1 C u	217-	D1. ..2 .	Oi												
240	D1. K.1 C u	220	D1. K.1 .	i												
241	A1. ..1 C u	119	... ..1 .	oi	121	... A.1 .	Goi	122	... K.1 .	Goi	243	A1. A.1 C G	244	A1. K.1 C G		
242	A1. C.1 C u	135	... ..1 C	Go	136	... C.1 C	o	241	A1. ..1 C	G						
243	A1. A.1 C u	119-	... ..2 .	oOi												
244	A1. K.1 C u	122	... K.1 .	oi												
245	K1. ..1 C u	247	K1. A.1 C	G	246	K1. K.1 C	G	287'	K.1 ..1 .	oi	291'	K.1 A.1 . Goi	292	K.1 K.1 . Goi		
246	K1. C.1 C u	245	K1. ..1 C	G	287'	K.1 ..1 .	Goi	289'	K.1 C.1 .	oi						
247	K1. A.1 C u	287-	K.1 ..2 .	oOi												
248	K1. K.1 C u	292	K.1 K.1 .	oi												
249	.1. ..1 D	250	.1. C.1 D	G	251	.1. A.1 D	G	252	.1. K.1 D	G	253	C1. ..1 D g	254	C1. C.1 D gG	255	C1. A.1 D gG
		256	C1. K.1 D	gG	261	A1. ..1 D	g	262	A1. C.1 D	gG	263	A1. A.1 D gG	264	A1. K.1 D gG	265	K1. ..1 D g
		266	K1. C.1 D	gG	267	K1. A.1 D	gG	268	K1. K.1 D	gG						
250	.1. C.1 D	249	.1. ..1 D	G	253	C1. ..1 D	gG	254	C1. C.1 D	g	261	A1. ..1 D gG	262	A1. C.1 D g	265	K1. ..1 D gG
		266	K1. C.1 D	g												
251	.1. A.1 D u	249-	.1. ..2 D	O	261-	A1. ..2 D	gO	263	A1. A.1 D	g	265-	K1. ..2 D gO	267	K1. A.1 D g		
252	.1. K.1 D	256	C1. K.1 D	g	264	A1. K.1 D	g	268	K1. K.1 D	g						
253	C1. ..1 D	249	.1. ..1 D	g	250	.1. C.1 D	gG	251	.1. A.1 D	gG	252	.1. K.1 D gG	254	C1. C.1 D G	255	C1. A.1 D G
		256	C1. K.1 D	G	257	D1. ..1 D	o	258	D1. C.1 D	Go	259	D1. A.1 D Go	260	D1. K.1 D Go		
254	C1. C.1 D	249	.1. ..1 D	gG	250	.1. C.1 D	g	253	C1. ..1 D	G	257	D1. ..1 D Go	258	D1. C.1 D o		
255	C1. A.1 D u	249-	.1. ..2 D	gO	251	.1. A.1 D	g	253-	C1. ..2 D	O						
256	C1. K.1 D	252	.1. K.1 D	g	260	D1. K.1 D	o									
257	D1. ..1 D	249	.1. ..1 D	g	250	.1. C.1 D	gG	251	.1. A.1 D	gG	252	.1. K.1 D gG	258	D1. C.1 D G	259	D1. A.1 D G
		260	D1. K.1 D	G												
258	D1. C.1 D	249	.1. ..1 D	gG	250	.1. C.1 D	g	257	D1. ..1 D	G						
259	D1. A.1 D u	257-	D1. ..2 D	O												
260	D1. K.1 D	252	.1. K.1 D	g												
261	A1. ..1 D u	151	... ..1 D	o	153	... A.1 D	Go	154	... K.1 D	Go	263	A1. A.1 D G	264	A1. K.1 D G		
262	A1. C.1 D u	151	... ..1 D	Go	152	... C.1 D	o	261	A1. ..1 D	G						
263	A1. A.1 D u	151-	... ..2 D	oO												
264	A1. K.1 D u	154	... K.1 D	o												
265	K1. ..1 D u	267	K1. A.1 D	G	268	K1. K.1 D	G	293'	K.1 ..1 D	o	295'	K.1 A.1 D Go	296	K.1 K.1 D Go		
266	K1. C.1 D u	265	K1. ..1 D	G	293'	K.1 ..1 D	Go	294'	K.1 C.1 D	o						
267	K1. A.1 D u	293-	K.1 ..2 D	oO												
268	K1. K.1 D u	303	K.1 K.1 K	oi												
269	.1. ..1 A u	209	.1. ..1 .	p	211	.1. A.1 .	Gp	212	.1. K.1 .	Gp	221	A1. ..1 . gp	223	A1. A.1 . gGp	224	A1. K.1 . gGp
		225	K1. ..1 .	gp	227	K1. A.1 .	gGp	228	K1. K.1 .	gGp	270	.1. A.1 A G	271	.1. K.1 A G	278	A1. ..1 A g
		279	A1. A.1 A	gG	280	A1. K.1 A	gG	281	K1. ..1 A	g	282	K1. A.1 A gG	283	K1. K.1 A gG		
270	.1. A.1 A u	209-	.1. ..2 .	Op	211	.1. A.1 .	p	221-	A1. ..2 .	gOp	223	A1. A.1 . gp	225-	K1. ..2 . gOp	227	K1. A.1 . gp
		269-	.1. ..2 A	O	278-	A1. ..2 A	gO	279	A1. A.1 A	g	281-	K1. ..2 A gO	282	K1. A.1 A g		
271	.1. K.1 A u	212	.1. K.1 .	p	224	A1. K.1 .	gp	228	K1. K.1 .	gp	280	A1. K.1 A g	283	K1. K.1 A g		
272	C1. ..1 A u	209	.1. ..1 .	gp	211	.1. A.1 .	gGp	212	.1. K.1 .	gGp	213	C1. ..1 . p	215	C1. A.1 . Gp	216	C1. K.1 . Gp
		269	.1. ..1 A	g	270	.1. A.1 A	gG	271	.1. K.1 A	gG	273	C1. A.1 A G	274	C1. K.1 A G		
273	C1. A.1 A u	209-	.1. ..2 .	gOp	211	.1. A.1 .	gp	213-	C1. ..2 .	Op	215	C1. A.1 . p	269-	.1. ..2 A gO	270	.1. A.1 A g
		272-	C1. ..2 A	O												
274	C1. K.1 A u	212	.1. K.1 .	gp	216	C1. K.1 .	p	271	.1. K.1 A	g						
275	D1. ..1 A u	217	D1. ..1 .	p	219	D1. A.1 .	Gp	220	D1. K.1 .	Gp	276	D1. A.1 A G	277	D1. K.1 A G		
276	D1. A.1 A u	217-	D1. ..2 .	Op	219	D1. A.1 .	p	275-	D1. ..2 A	O						
277	D1. K.1 A u	220	D1. K.1 .	p												
278	A1. ..1 A u	119	... ..1 .	op	121	... A.1 .	Gop	122	... K.1 .	Gop	167	... ..1 A o	168	... A.1 A Go	169	... K.1 A Go
		221	A1. ..1 .	p	223	A1. A.1 .	Gp	224	A1. K.1 .	Gp	279	A1. A.1 A G	280	A1. K.1 A G		
279	A1. A.1 A u	119-	... ..2 .	oOp	167-	... ..2 A	oO	223	A1. A.1 .	p						
280	A1. K.1 A u	122	... K.1 .	op	169	... K.1 A	o	224	A1. K.1 .	p						

nr	state	nr	state	why	nr	state	why	nr	state	why			
281	K1. ..1 A u	225	K1. ..1 .	p	227	K1. A.1 .	Gp	228	K1. K.1 .	Gp	282 K1. A.1 A G	283 K1. K.1 A G	287' K.1 ..1 . op
"		291'	K.1 A.1 .	Gop	292	K.1 K.1 .	Gop	299'	K.1 ..1 A	o	301' K.1 A.1 A Go	302 K.1 K.1 A Go	
282	K1. A.1 A u	227	K1. A.1 .	p	287÷	K.1 ..2 .	o0p	299÷	K.1 ..2 A	o0			
283	K1. K.1 A u	228	K1. K.1 .	p	292	K.1 K.1 .	op	302	K.1 K.1 A	o			
284	..1 ..1 . u	30	... ... A	i	32"	... A.. A	Gi	33"	... K.. A	Gi	36 A.. A.. A gGi	37" A.. K.. A gGi	38 K.. K.. A gGi
"		286"	..1 A.1 .	G	287"	..1 K.1 .	G	290	A.1 A.1 .	gG	291" A.1 K.1 . gG	292 K.1 K.1 . gG	
285	..1 C.1 . u	30	... ... A	Gi	31	... C.. A	i	32'	A.. ... A	gGi	33' K.. ... A gGi	34' A.. C.. A gi	35' K.. C.. A gi
"		284	..1 ..1 .	G	286'	A.1 ..1 .	gG	287'	K.1 ..1 .	gG	288' A.1 C.1 . g	289' K.1 C.1 . g	
286	..1 A.1 . u	167	... ..1 A	0i	173	A.. ..1 A	g0i	176	K.. ..1 A	g0i	290 A.1 A.1 . g	291' K.1 A.1 . g	
287	..1 K.1 . u	169	... K.1 A	i	175	A.. K.1 A	gi	178	K.. K.1 A	gi	291 A.1 K.1 . g	292 K.1 K.1 . g	
288	C.1 A.1 . u	167	... ..1 A	g0i	170	C.. ..1 A	0i	286	..1 A.1 .	g			
289	C.1 K.1 . u	169	... K.1 A	gi	172	C.. K.1 A	i	287	..1 K.1 .	g			
290	A.1 A.1 . u	297	..1 ..1 A	o0i									
291	A.1 K.1 . u	299	..1 K.1 A	oi									
292	K.1 K.1 . u	303	K.1 K.1 K	i									
293	..1 K.1 D	169	... K.1 A	i	172	C.. K.1 A	gi	175	A.. K.1 A	gi	178 K.. K.1 A gi	294 C.1 K.1 D g	295 A.1 K.1 D g
"		296	K.1 K.1 D	g									
294	C.1 K.1 D	169	... K.1 A	gi	172	C.. K.1 A	i	293	..1 K.1 D	g			
295	A.1 K.1 D u	299	..1 K.1 A	oi									
296	K.1 K.1 D	303	K.1 K.1 K	i									
297	..1 ..1 A u	284	..1 ..1 .	p	286"	..1 A.1 .	Gp	287"	..1 K.1 .	Gp	290 A.1 A.1 . gGp	291" A.1 K.1 . gGp	292 K.1 K.1 . gGp
"		298"	..1 A.1 A	G	299"	..1 K.1 A	G	300	A.1 A.1 A	gG	301" A.1 K.1 A gG	302 K.1 K.1 A gG	
298	..1 A.1 A u	284-	..1 ..2 .	0p	286=	..1 A.1 .	p	287÷	K.1 ..2 .	g0p	290 A.1 A.1 . gp	291' K.1 A.1 . gp	297- ..1 ..2 A 0
"		298÷	A.1 ..2 A	g0	299÷	K.1 ..2 A	g0	300	A.1 A.1 A	g	301' K.1 A.1 A g		
299	..1 K.1 A u	287	..1 K.1 .	p	291	A.1 K.1 .	gp	292	K.1 K.1 .	gp	301 A.1 K.1 A g	302 K.1 K.1 A g	
300	A.1 A.1 A u	284-	..2 ..2 .	o0p	290	A.1 A.1 .	p	297-	..2 ..2 A	o0			
301	A.1 K.1 A u	287-	..2 K.1 .	op	291	A.1 K.1 .	p	299-	..2 K.1 A	o			
302	K.1 K.1 A u	292	K.1 K.1 .	p									
303	K.1 K.1 K												

304 states (out of 3125), 1619 transitions.

I

D520

EN

D52081 8

END