RICE UNIVERSITY

ITERATIVELY STRUCTURED INFORMATION PROCESSORS

by

Gerald Randolph Kane

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

Thesis Director's Signature

Houston, Texas

May 1972
Copyright
Gerald Randolph Kane
1972
Abstract

Gerald Randolph Kane

ITERATIVELY STRUCTURED INFORMATION PROCESSORS

There exists a widely held belief that the full potential of LSI technology will be realized by arrays of logic circuits. A novel machine architecture is proposed that consists of a uniform array of identical cells with the property that the array executes a high-level programming language directly. The central idea of this thesis is that it is possible to exploit the geometry of an iterative machine architecture to achieve some degree of efficiency when dealing with languages that are structure-oriented. To this end, synthesis techniques for such languages (machines) are developed. Some of the more important features of such machines are their inherent abilities to sustain concurrent processes, to obtain a high degree of reliability through dynamic fault analysis, and to maintain efficient information storage. A detailed development and simulation of a language-machine in this architectural style is presented.
For my wife, without whose help this dissertation would have been impossible and without whose love it would not have been worthwhile.
Contents

I  An Introduction to the Concept of Iteratively Structured Information Processors 1

II A Partial Collection of Existing Philosophies of Structure-Oriented Programming Languages and Iteratively Structured Machine Organization 33

III Synthesis Techniques for Iteratively Structured Information Processors 73

IV Theoretical Bases for Iterative Language Machines 118

V Summary, Conclusion and Extensions for Machines of the Iterative Architectural Style 140

Appendices

A Annotated Bibliography of Material Related to the Design of Iteratively Structured Language Processors 146

B ALP - a List Processing Language to Embed in ALGOL 155

C Asynchronous Operation of an LCLM 163

D LISP-like Structure in a GPM Environment 172
<table>
<thead>
<tr>
<th>Illustrations</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 A Turing Multiplier</td>
<td>8</td>
</tr>
<tr>
<td>1.2 Garbage Collector -Isolation Phase</td>
<td>16</td>
</tr>
<tr>
<td>1.3 Garbage Collector -Compactor Phase</td>
<td>17</td>
</tr>
<tr>
<td>1.4 Locating a String Named A in a Lee Cell Memory</td>
<td>20</td>
</tr>
<tr>
<td>1.5 Execution of a Very Tight Loop</td>
<td>26</td>
</tr>
<tr>
<td>1.6 Set - To Sequence</td>
<td>28</td>
</tr>
<tr>
<td>2.1 Data Structures for L^6</td>
<td>36</td>
</tr>
<tr>
<td>2.2 Interpretive Cycle of IPL V</td>
<td>38</td>
</tr>
<tr>
<td>2.3 Data Structure for LISP</td>
<td>45</td>
</tr>
<tr>
<td>2.4 Ring Structure Example for Directed Graphs</td>
<td>49</td>
</tr>
<tr>
<td>2.5 Cylinders- An Associative Ring Structure</td>
<td>51</td>
</tr>
<tr>
<td>2.6 Cutpoint Arrays</td>
<td>55</td>
</tr>
<tr>
<td>2.7 Cellular Logic-in-Memory Arrays</td>
<td>59</td>
</tr>
<tr>
<td>2.8 Unger's Machine Locating Lower Left-hand Corners</td>
<td>63</td>
</tr>
<tr>
<td>3.1 Cell-Bus Interconnections</td>
<td>75</td>
</tr>
<tr>
<td>3.2 Control Bus Terminations</td>
<td>79</td>
</tr>
<tr>
<td>3.3 Simulation of the S^2P Machine</td>
<td>85</td>
</tr>
<tr>
<td>3.4 Simple Substitution Macros</td>
<td>99</td>
</tr>
</tbody>
</table>
3.5  Unilateral Asynchronous Bus Network
3.6  Double D Flip-Flop
3.7  Bus Register for an ISIP
3.8  Bilateral Asynchronous Bus Network
3.9  Stacking Mechanism in a Linear Iterative Machine
3.10 Enhanced Macro Facility

4.1  Garbage Collector with Bus Disturbance from the Left
4.2  Garbage Collector with Bus Disturbance from the Right
4.3  Simulation of the Garbage Collector in the \( S^2P \) Machine
4.4  Array Topology
4.5  Multiple Processes in an Iteratively Structured Information Processor

B.1  The Format of the ALP List System
Tables

1.1 Syntax for $S^2P$ 4
1.2 Garbage Signals Right 14
1.3 Garbage Signals Left 14
1.4 Cell Function by Contents for the $S^2P$ Iterative Machine 21

2.1 Axle Program Tables 46
Chapter I

An Introduction to the Concept of
Iteratively Structured Information Processors

For some time a widely held belief has existed that integrated circuit technology will increase to the point that large iterative arrays of logic circuits will become practical. To this end, many cell specifications have been proposed[1.1], and synthesis and analysis techniques have been developed for specifying these arrays. There is as yet no large rush to provided these circuits or to incorporate them in designs. Perhaps there are two reasons for this:

(1) synthesis techniques place a large premium on detail-- the basic structure of the designs are often obscured by the array structure; and (2) the customizing of large arrays as a final manufacturing step is still quite costly.

Cell complexity separates cellular logic into two not-so-distinct groups. The term micro-cellular is attached to those logic arrays whose basic cell configurations are on the order of complexity of a J-K flip-flop. These cells are often combinatorial and
the arrays employ micro-program techniques to realize various functions. The macro-cellular arrays have more complex cell structures. It is doubtful that micro-cellular logic arrays will achieve any degree of acceptance before macro-cellular representations become effective. This principle can best be stated--Complex cells are complex and simple cells are complex, too. The macro-cellular representations have the advantages that cell functions are more readily assimilated on a large scale and array customization through either manufacturing processes or stored program techniques are not an essential part of the design. It is entirely reasonable to project micro-cellular realizations of the macro-cells thereby achieving a high degree of iteration in the final analysis.

It would seem reasonable that cellular information processors could be constructed to execute higher-level languages directly.

For LSI to pay off, a new philosophy is needed for integrating software and hardware using greatly increased quantities of electronics to reduce total costs. [1.2]

Combining the system hardware and software into one unit should go a long way toward reducing computing
complexity and cost. The macro-cellular array appears to be a most reasonable way of achieving this end. The central idea of this thesis is that it is possible to exploit the geometry of an iterative machine architecture (its structure) to achieve some degree of efficiency when dealing with languages that are structure-oriented. Additionally, techniques for realizing iteratively structured machines may be useful in subsections of more conventional architectures.

The remainder of this chapter is concerned with the description of a very simple string language and an iterative machine to realize it. The language will serve as a base on which to introduce techniques for iterative machine synthesis (Chapter III). Chapter II is a survey chapter directed toward list processing languages, well-known iterative synthesis techniques and language based machines. The remaining chapter is devoted to determining specifications for machines to realize any general list processing language (Chapter IV).

The simple information processing language described here is a modified version of LINEAR LISP[1.3], a string rather than a list processing language. The list processing languages often deal
Table 1.1
Syntax for $S^2p$

A slightly modified form of B.N.F. is used to describe the syntax. The angle brackets <> are used to delimit meta-linguistic variables. The vertical stroke | is used to separate alternatives and square brackets [] are used to indicate repeated occurrences of the form they contain. The superscript ° is used to indicate an optional item that may be repeated zero or many times.

<ALPHANUMERIC>::= 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| | l | 2|3|4|5|6|7|8|9|0
<SYMBOLS>::=<ALPHANUMERIC>|cat|head|tail|goto|
        if|equals|then|end|;
<NAME>::=[<ALPHANUMERIC>]
<label<NAME>]:
<STATEMENT>::=<GOTO>|<ASSIGN>|<IF>
<STATEMENT LIST>::=<STATEMENT>;<<STATEMENT>>
°
<GOTO>::=goto<NAME>
<ASSIGN>::=set<NAME>to<EXPRESSION>
<IF>::=if<EXPRESSION>equals<TERM>then
<STATEMENT LIST>end
<TERM> ::= <NAME> | 'STRING'

<EXPRESSION> ::= <TERM> | <EXPRESSION>head |
<EXPRESSION>tail | <EXPRESSION>cat <EXPRESSION>

<NAMED STRING> ::= <LABEL><STRING>*

<PROGRAM> ::= [<LABEL><STATEMENT LIST>*] [NAMED STRING]*
with binary trees. A string is a binary tree where every left link points to a symbol and each right link points to another string. This linear structure makes it possible to simplify the addressing mechanisms of string languages. The syntax for the language $S^2P$, Simple String Processor, is given in Table 1.1. The semantics of the language are for the most part self-explanatory. The program consists of a set of strings identified uniquely by name. Strings may be decomposed in two ways by the use of the suffix operators $\text{head}$ and $\text{tail}$:

\begin{verbatim}
label ABC:XYZ#
ABC head=X
ABC tail=YZ
\end{verbatim}

and larger strings may be formed by catenation:

\begin{verbatim}
ABC cat ABC=XYZXYZ
ABC tail cat ABC head=YZX.
\end{verbatim}

The replacement or definition of a string may be effected by use of the assign statement. The only relational operator provided is the equal sign. It is the function of this operator to compare two strings of arbitrary length. The if-then construct is obvious. Control transfer to a named string is by way of the goto statement. These three simple statement types -- assign, if, and goto-- comprise the language.
A sample program is provided to illustrate the versatility and generality of the language by simulating a Turing machine[1.4]. Figure 1.1 illustrates a simple Turing machine and provides a directed graph representation for the machine. The chosen machine multiplies two unary numbers. The algorithm like most programs for Turing machines provides voluminous amounts of busy work to keep the machine occupied. The Turing machine program models the Turing tape as three strings LEFT, RIGHT, and SYMBOL (the symbol under the read head). Each of the state-symbol pairs has an associated labeled statement that provides the next state, print, and direction of motion information. The machine has four states and five symbols in its alphabet. The program is not extremely efficient but the power of the language should be evident.

Methods of implementing a simple language like this on conventional machines are well-known[1.5]; the reader is invited to recall such methods and contrast them with the iteratively structured approach. Consider now a linear array of cells each containing a single symbol. Strings can be formed by storing the symbols sequentially. A special area (the idea that
A) Turing Machine with initial tape configuration for multiplying 2 by 3

B) State transition graph — Edge labels represent read symbol; print symbol if different from read symbol/direction of motion

C) Halting configuration of tape

**Figure 1.1** A Turing Multiplier
A Program in $S^2P$ to Simulate a Turing Machine

'INITIATE SEQUENCE'

\[
\text{label START: set STATE to 'Q0'; goto INTERPRET;#}
\]

'MOVE TAPE'

\[
\text{label MOVE RIGHT: set LEFT to SYMBOL at LEFT;}
\]
\[
\text{set SYMBOL to RIGHT head;}
\]
\[
\text{set RIGHT to RIGHT tail;}
\]
\[
\text{goto INTERPRET;#}
\]

\[
\text{label MOVE LEFT: set RIGHT to SYMBOL at RIGHT;}
\]
\[
\text{set SYMBOL to LEFT head;}
\]
\[
\text{set LEFT to LEFT tail;#}
\]

'GROW TAPE'

\[
\text{label INTERPRET: if SYMBOL equals ' then set SYMBOL to '0';}
\]
\[
\text{set BRANCH to ' goto ' cat STATE cat';;}
\]
\[
\text{goto BRANCH;#}
\]

'TURING TAPE'

\[
\text{label LEFT: 11A#}
\]
\[
\text{abel SYMBOL: B#}
\]
\[
\text{label RIGHT: 111A#}
\]
'STATE 0'
labelQ00:gotoMOVERIGHT;
labelQ01:setSYMBOLto'0';setStateto'Q1';
gotoMOVERIGHT;
labelQ0A:gotoHALT;
labelQ0B:labelQ0X:gotoMOVELEFT;

'STATE 1'
labelQ10:labelQ11:labelQ1B:gotoMOVERIGHT;
labelQ1A:setStateto'Q2';gotoMOVELEFT;
labelQ1X:setSYMBOLto'1';gotoMOVERIGHT;

'STATE 2'
labelQ20:labelQ2A:labelQ2X:gotoMOVELEFT;
labelQ21:setSYMBOLto'X';setStateto'Q3';
gotoMOVERIGHT;
labelQ2B:setStateto'Q0';gotoMOVELEFT;

'STATE 3'
labelQ30:setSYMBOLto'X';setStateto'Q2';
gotoMOVELEFT;
labelQ31:labelQ3A:labelQ3B:labelQ3X:gotoMOVERIGHT;
such an area is a fixed portion of the array should be rejected as this violates the uniform cellular constraint) is to be used for assembling expressions. Unused strings will conveniently be left nameless and a dynamic "garbage collection" feature will be provided to keep the array tidy by removing unnamed strings. A good garbage collector is a necessity because this implementation produces many temporary strings; in fact the garbage collector is of such importance that its operation will be discussed first.

The garbage collecting algorithm consists of locating all unnamed strings in the array and erasing them. Named strings are then moved to the left so that in a completely garbage collected arrangement only named strings appear on the left and empty cells appear on the right. It is important to realize that the garbage collector and the string processor operate concurrently. The language is designed so that these two processes do not interact; there can be no conflict of interest so that both operate at the same level of priority. Because the string processor generates garbage as the garbage collector is identifying and removing the unwanted strings, an arrangement of the array where all the empty cells are together is possible only after the string processor has terminated
and the garbage collector has been permitted to complete its housekeeping chores.

Hennie[1.4] indicates that any regular expression can be realized by a linear array of cells with information flow in only one direction. The algorithm involves the synthesis of the appropriate sequential machine and transmitting state information to each succeeding cell with the input character as the cell location within the string. In this way time is replaced by spatial separation. Such a realization is inadequate for the proposed garbage collector for not only is it necessary to recognize the end of a suitable expression but its extent as well; thus bilateral flow is necessary in the garbage collection portion of the cells. Referring to the syntax of Table 1.1, a named string <NAMED STRING> consists of

\[
\text{label } <\text{name}> : <\text{string}> \#
\]

A simple regular expression for a string is then

\[
\text{ABB*CD*E}
\]

where

\[
\begin{align*}
\text{A} &= \text{label} \\
\text{B} &= \text{alphanumeric} \\
\text{C} &= : \\
\text{D} &= \text{any symbol and} \\
\text{E} &= \text{the string terminator} \#
\end{align*}
\]
The : is the primary symbol in identifying the named string. The string is just those symbols next to the colon with appropriate boundary symbols. From the right of the colon the regular expression
\[D^*E\]
describes the string and looking to the left
\[B^*BA\]
describes the string.

The cells each contain a single symbol and garbage collector signals are allowed to propagate to both the left(s) and right(r). It is expedient to assume that all the unused cells contain the end symbol \#. It is clear from the regular expression that signal flow from the right need carry only three kinds of information: that there is a termination cell (1) or a colon cell (2) or a cell containing any other symbol (3) to the right. The labeling of the input (incident) signals appears on the right in Table 1.2 in order to accentuate the "from the right" nature of the signal flow. From the left it is necessary to identify those cells following a colon up to and including the first \# as is shown in Table 1.3.

The initial signals for the cells depend solely on their contents, for example \(s_0\) for the string terminator \#, \(s_2\) for the colon, and \(s_1\) for any
Table 1.2
Garbage Signals from the Right

<table>
<thead>
<tr>
<th>$r_2$</th>
<th>$r_0$</th>
<th>$r_0$</th>
<th>$r_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_2$</td>
<td>$r_1$</td>
<td>$r_0$</td>
<td>$r_1$</td>
</tr>
<tr>
<td>$r_2$</td>
<td>$r_2$</td>
<td>$r_0$</td>
<td>$r_2$</td>
</tr>
</tbody>
</table>

Table 1.3
Garbage Signals from the Left

<table>
<thead>
<tr>
<th>Incident</th>
<th>$s_0$</th>
<th>$s_2$</th>
<th>$s_0$</th>
<th>$s_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_1$</td>
<td>$s_0$</td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td></td>
</tr>
<tr>
<td>$s_3$</td>
<td>$s_2$</td>
<td>$s_0$</td>
<td>$s_0$</td>
<td></td>
</tr>
</tbody>
</table>
other symbols. If at any iteration a cell input pair is $s_0r_0$ that cell is garbage and is erased except when the cell is a colon cell (this cell is, of course, not garbage and must be preserved). Figure 1.2 is provided to illustrate the action of the isolation phase of the garbage collector. Each row represents a successive instant of time. It is important to notice that the number of time steps required to isolate an unused string is equal to the length of the string. The cell containing B was identified as garbage at the second step. In general the garbage isolator identifies the center of an unused string of length $n$ in \( \lceil n/2 \rceil \) time steps and identifies the entire string in $n$ garbage collection cycles.

The compaction phase of garbage collection is illustrated in Figure 1.3. After sufficient garbage isolation steps the situation will arise that two named strings will be separated by more than one \# cell. The cell function must include the facilities for the forced migration of named cells to the left. A \# cell with $s_3$ as a left incident signal is changed to a special empty cell $\emptyset$ which acts as a vacuum to pull cells from the right. The cells containing $\emptyset$ do not modify either the right or left incident signals and may not be propagated to the left. Therefore only
FIGURE 1.2

GARBAGE COLLECTOR--ISOLATION PHASE
every other garbage collection cycle in the compactor phase can produce $\emptyset$ cells; this restriction is realized by not allowing the production of $\emptyset$ cells if either neighbor cell is a $\emptyset$ cell. In Figure 1.3 vertical displacement is used to separate time instants. A cell with $s_3$ incident will produce $\emptyset$ cells until a non-$\#$ symbol is adjacent to it. Once initiated the process of symbol migration achieves the maximum rate of one position per cycle. This is evident in the skewing of symbols along a minor diagonal.

The operation of the processor is best understood after a brief description of the Lee intercommunicating cells[1.7]. Each of the Lee cells contains memory for a symbol plus an activity bit and sufficient logic to interpret commands given the memory by a control element. These commands include

- **Reset**: Set all activity bits to 0
- **Right**: Propagate activity bits right
- **Match**: All cells with matching activity/symbol set their activity bits to 1 otherwise reset.

Figure 1.4 indicates the required actions for locating a string named $A$ in the Lee memory. First the array is reset. Then all the labels are identified by the match request $0/\text{label}$. The activity bits are passed right
and a match request for 1/A is made. This cycle of propagate right, match symbol is repeated until the name is exhausted. A propagate right followed by a match request 1/: will tag with a one activity all those strings (there may be more than one) with the desired name. In Figure 1.4 this is the string A. The proposed processor differs from the Lee memory in two important respects: (1) the cells are able to reflect different degrees of activity and (2) the sequential commands that direct the memory come from within the memory itself as opposed to an external director.

Commands are transmitted to all cells via the complex signal bus that carries information in the form of a complex symbol consisting of a signed activity and a symbol from either the language alphabet or one of the augmented symbols $\emptyset$, $u$, $\#$, or $\square$ which are used for special commands. The sequence of control is provided by a intra-cellular signal ($Z$) (which effectively modifies the state of the cell to the right). For the purposes of this discussion the array will operate under the control of a multiphase clock. The actions of all the cells are detailed in Table 1.4. The description here parallels the table and is intended to indicate how the resultant machine was derived.
Locating a string named A in a Lee cell memory.

**Figure 1.4**

Locating a string named A in a Lee cell memory.
Table 1.4

$S^2P$ Cell Specification Table

<table>
<thead>
<tr>
<th>Cell Type</th>
<th>Cell Act</th>
<th>Cell State</th>
<th>To bus</th>
<th>Bus Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>any</td>
<td>0</td>
<td>no</td>
<td>A(&lt;0) RLA;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A(&gt;0)  match/</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>match</td>
</tr>
<tr>
<td>2,3</td>
<td>0</td>
<td>no</td>
<td>contents RLA;</td>
<td>A</td>
</tr>
<tr>
<td>alpha</td>
<td>(&gt;0)</td>
<td>1</td>
<td>yes</td>
<td>contents RLA;</td>
</tr>
<tr>
<td></td>
<td>(\leq 0)</td>
<td>1</td>
<td>yes</td>
<td>0/\emptyset Cell act</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(Z\ rgt; RLA)</td>
</tr>
<tr>
<td>goto</td>
<td>0</td>
<td>1</td>
<td>yes</td>
<td>-1/\textbf{label} RLA;</td>
</tr>
<tr>
<td>set</td>
<td></td>
<td></td>
<td></td>
<td>(Z\ rgt)</td>
</tr>
<tr>
<td>;</td>
<td>1</td>
<td>1</td>
<td>yes</td>
<td>-0/# RLA</td>
</tr>
<tr>
<td>equals</td>
<td>2</td>
<td>1</td>
<td>yes</td>
<td>2/: (s_3,RLA)</td>
</tr>
<tr>
<td>then</td>
<td>0</td>
<td>3</td>
<td>yes</td>
<td>-0/# (s_4)</td>
</tr>
<tr>
<td>cat</td>
<td>0</td>
<td>4</td>
<td>no</td>
<td>not 0/\emptyset (s_3)</td>
</tr>
<tr>
<td>;</td>
<td>0</td>
<td>4</td>
<td>no</td>
<td>0/\emptyset RLA;0 act,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(Z\ rgt)</td>
</tr>
<tr>
<td>to</td>
<td>1</td>
<td>1</td>
<td>yes</td>
<td>-0/: (s_3)</td>
</tr>
<tr>
<td>if</td>
<td>0</td>
<td>1</td>
<td>yes</td>
<td>1/# (s_3)</td>
</tr>
<tr>
<td>to, cat</td>
<td>1</td>
<td>3</td>
<td>yes</td>
<td>-2/\textbf{label} RLA;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(Z\ rgt)</td>
</tr>
<tr>
<td>equals</td>
<td>0</td>
<td>4</td>
<td>no</td>
<td>0/\emptyset (s_3,SLA\ 2)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>yes</td>
<td>-3/\textbf{label} RLA;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(Z\ rgt)</td>
</tr>
<tr>
<td>Cell Type</td>
<td>Cell Act</td>
<td>Cell State</td>
<td>To bus</td>
<td>Bus</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>------------</td>
<td>--------</td>
<td>-----</td>
</tr>
<tr>
<td>then</td>
<td>3</td>
<td>yes</td>
<td>3:/</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>no</td>
<td>0/φ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>yes</td>
<td>-0/#</td>
<td>SLA 2,s₄</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>no</td>
<td>3/#</td>
<td>cell act, Z rgt; RLA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0/φ</td>
<td>RLA; -0 act, Z rgt</td>
</tr>
<tr>
<td>end</td>
<td>any</td>
<td>yes</td>
<td>0/φ</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>head</td>
<td>0</td>
<td>yes</td>
<td>-0/u</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>yes</td>
<td>2/φ</td>
<td>0 act, Z rgt</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>no</td>
<td>any</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>yes</td>
<td>2:/</td>
<td>RLA;s₁</td>
</tr>
<tr>
<td>tail</td>
<td>0</td>
<td>yes</td>
<td>2/u</td>
<td>RLA; 0 act, Z rgt</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>yes</td>
<td>2:/</td>
<td>RLA,s₁</td>
</tr>
<tr>
<td>cat</td>
<td>0</td>
<td>yes</td>
<td>-0/u</td>
<td></td>
</tr>
<tr>
<td>quote(')</td>
<td>-1</td>
<td>yes</td>
<td>0/φ</td>
<td>-1 act, Z rgt</td>
</tr>
<tr>
<td>label</td>
<td>1</td>
<td>0,1</td>
<td></td>
<td>0 act, Z rgt</td>
</tr>
<tr>
<td>quote</td>
<td>0,1</td>
<td>1</td>
<td></td>
<td>-2 act, Z rgt</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>-2</td>
<td>2 act, Z rgt</td>
</tr>
<tr>
<td>colon(:)</td>
<td>1</td>
<td>0</td>
<td>no</td>
<td>-0/#</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-0:/</td>
</tr>
</tbody>
</table>
Table 1.4 concluded

<table>
<thead>
<tr>
<th>Cell Type</th>
<th>Cell Act</th>
<th>Cell State</th>
<th>To bus</th>
<th>Bus Act</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>□</td>
<td>any</td>
<td>0</td>
<td>no</td>
<td>1/sym</td>
<td>copy; □ rgt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-0/≠</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-1/label</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-0/u</td>
<td>s₂</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2/sym</td>
<td>copy; □ rgt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#</td>
<td>any</td>
<td>0</td>
<td>no</td>
<td>-0/u</td>
<td>s₂</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sym rgt</td>
<td># rgt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>match</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>sym rgt</td>
<td># rgt</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>no match</td>
<td>cell sym to ≠</td>
</tr>
</tbody>
</table>

# 0 1 yes 0/φ if ≠ to right halt else 0 act, Z rgt

Legend:

RLA  reset local activity (cell activity to +0)
SLA  set local activity
rgt  pass parameters to right neighbor
A    bus activity
sym bus symbol
Z    direct right neighbor to state 1, reset cell state to 0

□ is equivalent to : with respect to the garbage collector

2/’ matches 2/: on the bus.
Any cell containing an alphanumeric symbol will output its positive activity and its symbol to the complex symbol bus and then propagate its activity and the Z signal right. All cells interpret the signal on the complex bus in the following manner: if the activity on the bus is zero, the condition is idle, i.e. a NOP command. If the activity is less than zero (-1, -2, -3) the cell symbol is compared to the bus symbol, and if a match occurs the cell propagates the absolute value of the bus activity to the right. For activity 1, 2, 3 both cell activity and symbol must match the complex bus symbol activity. The -0 is a cue for special operations which use the bus symbol to direct further action.

The action of these cells can be illustrated by a segment which branches to itself. The action of the two special cells goto and ; must be noted. First the cell containing 0/goto when in receipt of a Z signal will put -1/label on the bus, and propagate the 1 activity right and pass control to the right (Z signal). The Z signal incident on a 1/; triggers the following sequence: a -0/≠ to the complex bus, and no Z signal right. The -0/≠ is a special command that causes all 1/: cells to emit Z signals causing a branch in control. In Figure 1.5 a goto statement is just
starting execution. The -1/label is placed on the bus to locate all labels. Then the desired label "A" is found and the 1/; cell passes control to the desired location.

The set sequence requires a copy operation. The easiest way to construct a new string for the set operation is to copy the new name and end it with a colon and then copy the appropriate expression following the colon. This simple scheme will fail when the string to be replaced is involved in the expression or the replace string already exits. It is simply not enough to tack a colon at the end of the copied name. Two things must happen when a name has been copied: (1) a pseudo colon must be generated that can be converted to a real colon at the completion of the assign statement and (2) the 1/colon cell associated with the existing name must be activated so that it can be deleted at the end of the assign. A string without a colon will revert to the free storage by action of the garbage collector.

To affect the copy operation the special cell □ is introduced. The action of this cell is to copy the symbol from the complex bus and propagate the □ to the cell on the right. Details of the copy, which is restricted by the activity on the bus, are also
FIGURE 1.5
EXECUTION OF A VERY TIGHT LOOP
presented in Table 1.4.

The set cell function is exactly like the cell containing \texttt{goto}. The to cell in control first outputs -0/: to the bus in order to create the pseudo colon and then outputs -2/label to begin the search for the name in the replacement string. The set-to sequence is presented in Figure 1.6. It is necessary to observe three sections of the array: to the left the segment directing the operation, in the center the current string named "A" that will be replaced and to the right the copy section. These various sections are shown at four consecutive time instants.

The expression evaluation sequence is straightforward. The reason for making \texttt{head} and \texttt{tail} suffix operators will be evident from the implementation. Consider the simplest case where the expression is simply a name. When the semicolon (statement terminator) is reached the name will have been located and the cell to the immediate right of the name will be 2/:. The semicolon must provide signals to copy the string; the copy indirect action is described in the table. First 2/: to the bus to make the cell following the 2/: cell in memory two active. Then, -0/u alternating with data until the copy operation is complete.
**FIGURE 1.6**  
SET - TO SEQUENCE
The catenation operator works similarly. First the $2/*$ cell directs the copy of the named string and then begins a search for a new name. The quote operator necessitates modification of the above scheme. Z incident on $2/quote$ causes the cell to retain the two activity, now when the Z signal reaches the $0/$ cell, a copy operation is indicated but there is no colon to bypass so the ; and similarly the cat use the 0,2-activity to determine the necessity of skipping a colon. The process of emitting $2/:$ to the bus and changing the cell activity to $+0$ is later refered to as colon bypass. Z incident on $0/quote$ is an idle condition and thus a way to imbed comments in the memory although such a comment must contain a colon to avoid the garbage collector. Quote cells cannot pass non-positive activities thus the second quote of a string will terminate the quote process.

The suffix head operator is handled as follows: the activation signal conditionally yields the colon bypass followed by a $-0/u$ to copy one symbol and then a $2/\phi$ to the bus. The $\phi$ symbol matches nothing so this is a clear-activities instruction. The tail operator does a colon bypass as appropriate then outputs $2/u$ to the bus. $2/u$ is a propagate-2-activities-right as $u$ matches anything.
Activity 0 is sent to the right along with the Z signal upon completion of the operation. From the preceding discussion it is clear that composition of operators is possible -- the operators being applied in order closest to the name, e.g. ABC tail tail head will select the third character in string ABC whereas ABC head tail tail will be #.

The conditional statement is a non-essential but useful feature of the language. Another special symbol # must be used to affect the comparison of two strings. The idea is to build the expression represented before the equals and then compare it character by character with the term following. For the cell pair #,x (where x is any symbol) x must match the bus symbol with an activity 3 which follows -0/u on the bus. The -0/u is provided by the then cell in the same way that * and ; drive the copy operation. Otherwise the # is deleted. If a match occurs the # is propagated right. The command -0/# forces the #□ pair to emit a 3/# if the #□ combination exists. The -0/# also replaces □ with ≠□ thus returning any remaining comparant strings to the free space. If the strings are not equal, the # will have been deleted or the # and □ will be separated by one or more characters. In this case the then cell driving the compare will not receive
the affirmative $3/\#$ and will propagate -0 and Z to the right. The -0 activity inhibits further communication with the bus until the end cell is reached.

Some comments regarding garbage generation-collection rates are in order. Only 1-active cells generate garbage at the rate of one cell per cycle; 2,3-active names generate no garbage; and the copy operation generates garbage at $1/2$ cell per cycle. As there are certainly more class 2,3-names than those of class 1, the effective garbage generation rate is $1/2$ cell per cycle.

The garbage collector does not operate for the first $n/2$ time instants for a string of length $n$ but collects garbage at the rate of two cells per cycle thereafter giving a net rate of one cell per time instant. This is conveniently fast enough to stay ahead of the generation. Additionally the garbage collection process is an inherently faster one than the string process so that the memory processor is on the average neat and tidy.
References for Chapter I


[1.2] Rex Rice, "Impact of Arrays on Digital Systems," IEEETSSC IV.


Chapter II

A Partial Collection of Existing Philosophies of Structure-Oriented Programming Languages and Iteratively Structured Machine Organization

HOW TO READ THIS CHAPTER--If the reader is convinced that the ideas in Chapter I are worth pursuing then he may skip this chapter completely. If he is however uncertain that this represents a useful area, then Chapter II may provide additional motivation for the acceptance of this concept. Finally if the reader is a member of the author's thesis committee he may wish to read this chapter to determine that iteratively structured language machines are indeed a contribution to the state of the art.

Part 1: The Languages

The fundamental idea behind a large class of programming languages is that the relationship between individual data items or groups of items is often more important than the data itself. These languages exist to manipulate the structure of the data and do so quite often at the expense of efficient operation on the data. One can consider the structure information to be
contained in a list, hence the terms list processing and structure-oriented programming languages. The author prefers the latter term, although the former is more common.

The characteristic providing the largest separation between the various structure-oriented languages is the classification of the type of structure to be handled by these languages. The kinds of structures most often considered for list processing languages include linear lists, tree structures, and arbitrary structures such as directed graphs. It is the purpose of this section to discuss some of the common languages and those with novel features. These will be discussed by first examining some of the low-level languages and then by surveying the field by structure class.

When examining the various list processing languages, the features provided within a class are essentially the same, only the syntactic representation varies to be more applicable to a given situation. In perhaps no other programming experience is the argument for an extensible language base more pronounced. The disturbing fact that extensible languages lead to many incompatible variants of the same idea is offset somewhat because in list processing applications the
practitioners tend to create languages to serve their own ends anyway. The only language with a chance of providing unity is **LISP** which has a difficult syntax that people seem content to live with and complain about.

The language $L^6$ which is Bell Telephone Laboratory's **Low-Level Linked List Language**, was developed by Knowlton[2.1]. The user is able to write programs in this language at almost the assembly language level. The result is that very fast code can be generated although a certain amount of tedium is present. Storage is allocated in blocks of $2^n$ words thus making dynamic storage allocation simple. A free block of $2^n$ words may be split to form two free blocks of $2^{n-1}$ words and two contiguous unused blocks of $2^{n-1}$ words can be returned to the free space as a $2^n$ word block. Figure 2.1 diagrams the $L^6$ data structure. Within each word of a block the programmer may define fields using single character names that encompass any number of contiguous bits; these fields may then be used for any purpose. There are a number of base fields called bugs that are used to trace through the data structure. The referencing of field contents is done by concatenating field names as appropriate; as the field names are only one character
There are six user-defined fields: A and B are half word fields; C, E, and F are full words; the D field overlaps the low order end of the field C. Bugs W and X point to cells within the structure. The name BUG refers to the fact that these pointers traverse the structure and may be thought of as crawling about like tiny "varmints."
wide there is no need for special punctuation. The instruction format is very simple and consists of a test condition and a set of operations to be performed if the condition succeeds. The list of operators is extensive and facilities for arithmetic and input/output are provided. Recursive program capability is provided by three push-down lists that may contain field definitions, field contents, and subroutine linkages.

The push-down list is used to a larger degree in IPL V—Newell's Information Processing Language[2.2]. The central theme in IPL V is the specification of an assembly language for a list processing machine. The code is executed interpretively, and each interpretive cycle corresponds to one machine cycle (see Figure 2.2). The operation code structure of IPL resembles a two address instruction, one of which is used as the branch to the next instruction. There is also room for cell names (labels). Several addressing modes, including multi-level indirect addressing, are provided. The instruction format is:

\[
\text{NAME } P \ Q \ SYMB \ LINK
\]

where \(P\) is the operation code and \(Q\) the addressing mode. A large number of push-down lists are available to store intermediate program results. It is very easy
Figure 2.2
Interpretive Cycle of IPL V
for the program to generate more program; this adds greatly to the flexibility of the language. As an example, it is possible to use a program to generate efficient code (for the IPL machine) to recognize complex interrelationships in nonsense data. Recursive subroutines are easily effected, and the possibility exists for recursive/non-recursive calls of the same subroutine at the programmer's option.

The major drawback of IPL V is the existence of a very large number of system subroutines to do almost every imaginable task. This could be a very useful feature but each subroutine consists of a letter-number designation that is scarcely mnemonic, making impressive coding impossible to all except the highly experienced programmer and intelligible coding difficult for the novice.

The last of the low-level languages to be considered is NEUCLEOL[2.3], a primitive version, just the neucleus, of EOL. EOL (Expression Oriented Language)[2.4], a string processing language developed by L. Lukaszewicz, is of little interest after discussion of NEUCLEOL because the languages differ only in syntactic niceties. NEUCLEOL is a low level language for manipulating strings. By the use of a special set of parentheses, '$(' and '$)'), a block
structure is created that gives these strings tree-like properties. The actual manipulation of these strings using fields and pointers is transparent to the user. This is a significant advance toward a higher-level language from L⁶ and IPL V. Each string has an attribute or state identifier and an associated scanner; the scanner is a unique form of string that "moves" through the object string copying and transposing substrings according to the instructions the scanner passes. The instructions themselves are special instances of strings. Several data types are recognizable, and a formal macro facility exists. A large set of punctuating symbols is realized by using the escape character '$' to extend a common character set making legibility quite low.

Two of the simpler string processing languages are TRAC and GPM. TRAC is a language designed to handle unstructured text interactively[2.5]; it is similar in concept to Strachey's GPM[2.6] (for General Purpose Macro-generator). GPM will be discussed here as it is the model for the macro facility of the iterative machine described in Chapter III.

GPM operates on an input string of characters, and by a very general substitution algorithm transforms it into an output string. A single stack is used to store
macro definitions and partially expanded macro-calls. A macro-call consist of an opening bracket '[]' followed by the macro name and the actual parameters and terminated with a matching ']'). The macros are expanded by substituting the actual parameters into the macro body as required. A very important aspect of GPM is that a macro call can occur anywhere within the string.

DEF is a system macro with two arguments: the first is the name of the macro and the second the text string comprising the macro. <> are string quotes.

[DEF, EQ, ~1, [DEF, ~1, *f*], [DEF, ~2, *t*]]

the call to the macro EQ defines two macros one named "~1" with the value *f* and the other "~2" with the value *t* and then calls the macro "~1". If the strings are unequal only one macro with the name "~1" will exist and it returns a false result whereas if the strings are equal "~2" will be expanded as the processor stacks macro definitions from left to right as it encounters them and the macro "~2" will be nearer the top of the stack.

COMIT is a very early attempt at a string processing language. The operations take the form of rules which are applied whenever possible. A string is
searched until a substring is found that matches the input form of the rule, and the substring is then transformed to the output form of the rule. The COMIT operation takes place on a work string consisting of the concatenation of several constituents; the constituents may have any number of subscripts or attributes that affect the application of the various rules. Recursive procedures may be handled by the use of pushdown shelves, but syntax limitations make such operations clumsy at best. Special rules to enable the rapid search through dictionary-type strings make COMIT valuable as an information retrieval language.

The string processing language SNOBOL is important for its pattern-matching feature. Additionally, SNOBOL-like syntax pervades the string processing language business—L^6 and AXLE to mention just two. The common operation in SNOBOL^4[2.7] consists of finding a pattern in a given string. The pattern may be simple, such as a single character or string of characters, or it may be quite complex consisting of alternative strings positioned within the string, such as the closest vowel not nearer than four characters from the end of a string. SNOBOL has facilities for doing arithmetic and conversion between strings and integers is automatic. Recursive procedures are easily
implemented. As SNOBOL is interpreted rather than compiled, program modification and generation are easily handled. SNOBOL eliminates specific input/output statements by letting certain identifiers serve special purposes. For example INPUT identifies the next card to be read. SNOBOL formatting is restricted to card-width instructions, but this problem is more of a system limitation and is not deeply embedded in the language.

As a concluding example of string processing languages, AXLE will be presented— not because the ideas are novel but because they are typical and because the author likes it. AXLE is an AXiomatic Language [2.8] for string transformations. Data in the form of a string of characters is transformed according to a table of axioms, called imperatives. As in SNOBOL an imperative is applied on the basis of a matching procedure that determines which pattern to use. The patterns themselves are defined in a table and are called assertions. Recursive definitions in the assertion table are handled routinely. The structure of the imperative table consists of a left side which is a pattern, a right side which is the transformed version of the requested pattern and a transfer label within the imperative table. Any of
these parts may be omitted. The resultant forms lead to test (no right side) and insertion (no left side) operations. A sample AXLE program is shown in Table 2.1 for translating infix to Polish notation.

The LISP language[2.9] is a formal mathematical language designed for symbolic manipulation. Because the language can be expressed precisely, LISP is found in a variety of applications, particularly artificial intelligence studies. The basic structure of LISP as indicated in Figure 2.3 is the binary tree. The branches of the tree are accessed by the primitives car (the head of the list) and cdr (the tail of the list). These mnemonics are significant acronyms on the machine originally used to realize LISP. The program structure of LISP differs significantly from other programming languages. In LISP functions are described that are then applied to the data structures as opposed to the sequential nature of most algorithmic languages. Recursive programming in LISP is thus a necessity. LISP programs are interpreted but may be compiled in order to speed computation. Storage management is accomplished by the wanted-list technique. As the LISP supervisor knows what lists are required, all the lists in use can be traced from a known set of pointers. When the available space is exhausted, the
FIGURE 2.3
DATA STRUCTURE FOR LISP
Table 2.1

Assertion Table

\begin{tabular}{c}
\hline
\( \circ = + \\
\hline
\( \circ = - \\
\hline
\( \circ = \times \\
\hline
\( \circ = \div \\
\hline
P = M \\
Q = M \\
M = A \\
M = B \\
M = C \\
M = D \\
M = \circ PQ \\
\hline
\end{tabular}

Imperative Table

\begin{tabular}{c}
1 \((P \circ Q) = \circ PQ\) 1 \\
2 \hspace{1cm} \text{END} \\
\hline
\end{tabular}

Example

\(((A+B)-C) \times D\) becomes \(\times + ABCD\)
garbage collector examines the wanted list area, collects those lists that are needed and returns the unused space to the free area.

All of the languages previously discussed were stand-alone languages; that is, they were complete languages and not designed to be embedded in another host language. There are several useful list processing languages that are designed as subroutines for another high-level language. Two examples of LISP imbedded in ALGOL are LITHP[2.10] (say it aloud) and ALP. LITHP mimicks LISP completely whereas ALP is a packet of procedures written by the author to exploit the macro facility available in Burroughs Extended ALGOL. A large number of the primitives in ALP may be compiled but these same primitives are procedure calls in LITHP. The garbage collection mechanism in ALP depends on the programmer releasing space when it is no longer needed. This unwanted-list mechanism is more difficult for the programmer but somewhat easier to implement. An ALP description is provided in Appendix B.

SLIP[2.11] is a list processing system intended for embedding in FORTRAN in which each cell has a forward and backward link. This feature makes it just as easy to access the end of a list as its beginning.
The useful features of having a host language such as FORTRAN are of course applicable to SLIP, e.g. FORTRAN arithmetic and I/O facilities are available. FORTRAN is however not recursive and SLIP must provide subroutines to accomplish the needed recursive features. This makes SLIP a rather large subroutine packet. Garbage collection in SLIP is similar to ALP (actually the reverse is true). The program frees a list by the IRAST subroutine. A reference count of the number of times a list is used as a sublist is kept within the list itself and when this count goes to zero the list is returned to the list of available space. Because both the beginning and end of a list are available (the doubly linked feature) garbage collection is simpler.

The actual data structure of SLIP more nearly resembles a ring rather than a tree. A ring is a singly linked list of pointers that is closed; i.e., the last element of the ring points to the first. Figure 2.4 depicts an example of a simple ring and a directed graph in a ring structure. The primary ring is a list of nodes of the graph and the subordinate rings are the links originating at a given node-- the graph represented is that of the Turing machine of Figure 1.1. The ring data structure is of particular
(A) A SIMPLE RING

(b) A DIRECTED GRAPH USING A RING STRUCTURE

FIGURE 2.4

RING STRUCTURE EXAMPLE FOR DIRECTED GRAPHS
interest in graphics processing where the data structures tend toward the complex. The GRASP[2.12] system is an example of a structure-oriented language using rings.

An enhanced ring structure not currently available in any common programming language is the cylinder[2.13]. The cylinder combines arrays and rings. Simple hardware mechanisms, such as index registers, are available for accessing data in the arrays. A cylinder consists of at least two rings linking arrays called seams. A simple cylinder and some possible sub-structures are illustrated in Figure 2.5. Compare the simplicity of the cylinder structure with the general ring structure of Figure 2.4.

The usefulness of directed graph-like structures demands that programming languages to deal with them be developed, but the complexity of the structure required has severely limited the development of such languages. Two graph-based languages will be discussed here. The first is GEA for Graph Extended ALGOL[2.14]. GEA as its name implies is an extension to ALGOL syntax to include graphs. GEA can be added to ALGOL by a pre-pass phase to compress the extended form to something recognizable by the existing compliers. The graph operations provided include union (combining two
FIGURE 2.5

CYLINDERS - AN ASSOCIATIVE RING STRUCTURE
graphs), intersection, complementation of a graph, and merging two nodes within a graph. Graphs are stored as two lists-- one containing the nodes of a graph and the other containing all nodes with a single link to another node. Links and nodes are identified by number. There is an extended for statement (forevery) for dealing with links and nodes in graphs.

GRASPE is a language for manipulating graphs that has been described in terms of its semantics[2.15]. It has no specific host language although LISP has been used for such a purpose. GRASPE is represented by a set of operations on a hypergraph. The hypergraph is the set of all graphs that exist in a computation at a given point. An example of such an operation is son(N,G) for set of outpointing nodes which returns a set of all the nodes of graph G that have a directed link from node N. There are operations to create, destroy, and test the elements of a desired graph. The utility of directed graphs is demonstrated in the Pratt-Friedman paper[2.15].

The essential features of all structure-based programming languages deal with the structures manipulated by the languages, the effective use of storage, the capabilities for recursive programming, and the possibilities of self-modification.
Part 2: Iterative Machines

The concept of cellular automata is of more than just historical interest to iterative circuit designers. The ideas concerning path building and access rules of Holland undoubtly have their bases in the cellular automata of Von Neumann. The cellular automata concept developed out of a desire to model complex evolutionary systems. As a consequence of this, two problems occupy a large place in the theory of cellular automata, namely the self-reproducing problem as dealt with by Von Neumann and an equally complex concept dealing with configurations of cells that cannot be generated by other configurations, the so-called Garden of Eden problem[2.16].

The Von Neumann self reproducing automaton has a cell of 29 states and the state transition function (neighborhood relation) is a function of the four adjacent squares. An interesting result due to the development of a game, LIFE, invented by Conway[2.17] has provided a particularly simple description of a cellular automaton with its eight nearest neighbors in its state function and only two states per cell! The two state-cell in the Von Neumann neighborhood is inadequate to construct a universal machine. The idea
here is to synthesize a Turing machine that copies itself in the cellular space.

The work of Hennie in iterative circuit analysis has been previously cited in Chapter I in dealing with the recognition of regular expressions in cellular arrays. Another interesting result provides that as the number of directions of information flow in an iterative network increases the number of classes of expressions that are realized increases also[2.18]. It is worth mentioning that not all of these classes may be worth recognizing.

More nearly in the realm of practical iterative circuit synthesis are the cutpoint cellular arrays of Minnick[2.19]. Minnick shows that a suitably restricted set of binary functions in a tailored array are sufficient to realize arbitrary switching functions of n variables. Two such arrays are illustrated in Figure 2.6, one from the original cutpoint paper to add one to a BCD digit and another representing the activity match portion of the iterative machine of Chapter I (that is, an activity match occurs if the bus activity is negative or the positive bus and cell activities are equal - match<~busactsgn ∨ (busact0=cellact0) ∧ busact1=cellact1). The important thing to realize is that the two arrays are
INDEX | FUNCTION (Z)
0 | $1$
1 | $\bar{Y}$
2 | $\bar{X} + \bar{Y}$
3 | $\bar{X} \bar{Y}$
4 | $X + Y$
5 | $X \bar{Y}$
6 | $X \odot Y$
7 | 0
13 | R-S FLIP-FLOP

**FIGURE 2.6 A**

CUTPOINT CELL
FIGURE 2.6 B

Add-one Circuit - Cutpoint Array
FIGURE 2.6 C

Activity Match in a Cutpoint Cellular Array
incomprehensible to all but the most rigorous of scrutiny. It is this unfortunate drawback that plagues micro-cellular design.

Huang[2.20] has provided an alternate approach to the realization of a set of functions in an array by utilizing a collection of autonomous sequential machines which are arranged as a set of rows with custom tailoring at the boundary. More complex cells and interconnection structures[2.21,2.22] exist and the uniformity and complexity detract from the understanding to an even greater degree. Minnick's cobweb cells are an example of this complexity dilemma; this very powerful construction technique fails to distinguish any function except under a magnifying glass.

This is not to say that all iterative micro-cellular arrays are nebulous. When the array in question performs some globally significant function then array specification is perhaps best. Consider the cellular sorting arrays of Kautz[2.23] and Kane[2.24] where the functions of the array may be globally assimilated. It is desired to store numerical data in the array on the basis of an order relation (Figure 2.7). The rows of such an array form one word comparators. In this case although the array is
FIGURE 2.7

CELLULAR LOGIC-IN-MEMORY ARRAYS

CELL EQUATIONS:
\[
\hat{x} = x(y + \bar{o}) \\
\hat{y} = y(x + q) \\
c = w\hat{x} \\
r = r\hat{x}
\]
cellular in nature the combined cell function is easily discernable at a global level.

A more detailed examination of micro-cellular techniques is beyond the intent of this chapter. For the most part then, the iterative machines discussed here are actually modular machines. The reader is reminded that the concern is with modular machines or subsystems with a uniform interconnecting structure. Modular systems such as the Washington University macromodular computer[2.25] which consists of a large set of similar modules and an irregular control structure do not satisfy the uniformity constraints. The addition of special circuitry at the boundary of an array does not constitute a violation of the uniform cellular regulation.

Holland has proposed an iterative circuit computer capable of executing an arbitrary number of subprocessess simultaneously[2.26]. Although the Holland machine is completely unprogrammable, the concepts employed are worth consideration. The processor is a uniform array of computing elements each of which contains an arithmetic unit, some memory, and the necessary control circuitry. The processor operates in two phases: a path-building phase and a computing phase. During the path-building phase a
system of interconnections (through the various cells) is established beginning at some arithmetic processor (arithmetic refers to the state of the cell not its permanent function) and terminating at at least one memory cell. During the computation phase a calculation is performed between the arithmetic process accumulator and the referenced memory location, or control is passed to that referenced memory cell(s) thus giving the possibility of initiating several subprocesses. Assembly-level programming for such a machine is out of the question, and no suitable compiler techniques exist for the Holland machine because it is not a proposal for a piece of hardware-- but a thought machine.

A variant of the Holland machine has been proposed by Comfort[2.27] that considers placing a set of arithmetic units (a-boxes) at the perimeter of an array of path-finders and memory elements. Calculations are performed by connecting the various cells with the correct a-boxes. This modification makes the cost of the array less because the cells are not quite so general in function, but the array can still not be programmed.

A final example of Holland-based machines is that of Gonzales[2.28]. His machine operates on three
planes-- a program layer, a control layer, and a computing layer in a kind of sandwich effect. The function of the control layer is to provide program access to the computing layer by building paths and doing "look-ahead" operations in conjunction with the program layer. Such a machine is well suited to the processing of data that have a spatial orientation. The problem of programability has not yet been solved.

The iterative machine of Unger[2.28] consists of a plane array of cells each of which contains a one bit accumulator and a small random-access memory-- six(6) one-bit words. The cells are under the direction of a central control which provides commands for the logical manipulation of single bits on a cell basis and the communication of cell information among the four nearest neighbors. This machine is particularly suited to processing data in a spatial format-- such as visual feature selection. Figure 2.8 shows the action of Unger's machine in locating lower left hand corners.

The SOLOMON computer[2.30] and its successor the Illiac IV[2.31] are two examples of array processors where individual cells are quite complex and few in number (at most a few hundred). The SOLOMON computer (Simultaneous Operation Linked Ordinal Modular Network) consists of an array of process elements
**Figure 2.8**

Unger’s Machine Locating Lower Left-hand Corners

(This version of lower left-hand corner locator due to Minnick.)
(PE) under the direction of a network control element. The PE's communicate with their four nearest neighbors and at the periphery with a number of I/O channels. The Illiac IV is a large faster machine of the SOLOMON style where array elements are fast general-purpose computers in their own right. By exploiting parallelism, machines of the Illiac IV class are undoubtedly among the fastest machines currently available. Unfortunately the kinds of problems such machines handle well are restricted to those that can best be handled on a spatially iterative basis such as partial differential equations.

A cellular computer has been designed to implement the Kalman filter algorithm[2.32]. The design makes use of an array structure to perform the many matrix and vector calculations required in that algorithm. The basic structure of the computer consists of a universal control element, a column register, a row register and the array itself. The universal control element directs the other three and handles those calculations not suitable to the matrix approach. Extensive use of data routing paths to affect the matrix operations is present. The machine itself is described in terms of a small set of building blocks. The utility of such an approach to specialized
calculations is evident; it is unclear whether the approach is useful in dealing with general programming languages (as in a cellular APL machine described in the next section).

The intercommunicating cells of Lee have already been described in Chapter I. By suitable programming, this memory may be accessed by name or by contents. Sturman[3.33,3.34] has designed an iterative machine based on the Lee cells; the actions of this iterative computer are primitive and consist of passing activities to the left and right and matching symbols on a bus either directly or by an indirect mechanism. Although programming at the assembly language is very primitive, it is reasonable to expect a decent compiler for such a machine. The ability to do arithmetic and simple subroutine linkage are lacking. Sturman has proposed an asynchronous mode of operation exploiting the transmission-line properties of the bus.

The distinguishing feature of practically all of the iterative machines is their preoccupation with detail. The structure masks utility rather than enhances it.

Part 3: Language Directed Machine Design
The subsection title was borrowed from a paper by
McKeeman[2.35] dealing with the desirability of unified language and machine design.

No matter how machine independent we wished our programming languages to be they have all been overwhelmingly sequential, arithmetic and random access memory oriented.

The obvious attack for programmers and hardware people together is to devise language that reflects what we want to do and how we want to do it (for instance, in parallel) and machine structures efficient in handling that language. Let us call this method "language directed computer design".

The discussion here is concerned with those machines that have as their machine language a high-level language. These machines are not numerous but the idea is certainly not a new one. Included in those properties that characterize higher-level languages are (1) the use of symbolic addressing and names; (2) the establishment of program structure and a limitation of scope; (3) program division into statements and functions rather than instructions.

At least two proposals for FORTRAN language machines exist[2.36][2.37]; their design philosophies are similar. During the load phase of the FORTRAN program a symbol table is built and the program is translated to a form recognizable to the machine. Such a transformation is one-to-one and does not represent a compile phase. Such machines are
particularly attractive for one-shot computing tasks--the kind that are usually handled on an interactive and interpreted basis.

The SYMBOL machine[2.38] is an instance of a language-machine designed as a unit. Some of the more impressive features of SYMBOL are its independence of compilation procedures, variable precision arithmetic, and dynamic storage allocation (virtual memory). EULER has been implemented in micro program form on the IBM 360/30[2.39]. The machine EULER is essentially a hard-wired compiler.

At least two distinct varieties of APL machines exist. An APL machine for executing compiled code has been proposed that employs an active (process capable) cellular memory[2.40]. The design falls short on total modularity and places some severe restrictions on vector and array sizes. The cellular approach (which is similar to that of Cannon[2.32]) is used to enhance the matrix computation capabilities that are essential to APL but the aspects dealing with the language itself are poor.

A second approach to APL is being developed at the University of California at Berkley[2.41] that uses a micro-programmed processor for the interpretation of APL. Such an approach shows the feasibility of
dedicating the hardware to a higher-level language.

Yet, all of the machines maintain a random access memory and fail to devote a large amount of the design to a modular or iterative approach. There is certainly a place in the development of computer architecture for machines with large associative addressing memories and higher-level language bases. This thesis will fill a portion of that space by describing a class of machines that execute rather than interpret higher-level languages.
References for Chapter II


The design of the $S^2P$ in Chapter I is impractical for several reasons. The lack of any kind of macro or function facilities is an obvious shortcoming. The dependency on a central clock and the operation of the entire array in "lockstep" is particularly naive. In realizing an iterative machine the design engineer must be cognizant of the fact that the propagation of information through the array takes a non-zero amount of time; the operation of a machine that waits for information to propagate throughout the array will be unnecessarily slow. A built-in cell delay to compensate for this propagation time will need to be either adjustable or fixed at the maximum delay interval to permit operation in a variety of machine configurations. Neither approach is particularly appealing. This chapter describes synthesis techniques for iteratively structured information processors of the kind introduced in Chapter I with the universal clocking restriction removed. The description and simulation of an unclocked $S^2P$ with additional methods for realizing recursive macro calls, arithmetic
operations, vector operations, and parallel processing is provided.

It is helpful to imagine the interconnecting structure of the asynchronous or unclocked $S^2P$ machine to be a simple multi-rail transmission line where the propagation time is directly proportional to the inter-cell distance. The cells on the bus are at fixed constant spacing and the cells themselves make decisions in zero time. Figure 3.1a illustrates such an arrangement. This is equivalent to saying that all the delays are lumped into the transmission characteristics of the bus. This kind of bus represents an arrangement that is just as impractical as the synchronous machine in Chapter I. The dual approach of lumping all the delays in the cells with no propagation delay between cells as illustrated in Figure 3.1b more nearly represents the approach taken in this development but is conceptually more difficult. The common properties associated with transmission lines such as characteristic impedance and impedance matching phenomena will be used in the early development. The reader is cautioned that these ideas are presented because they are well known and easily understood; the realization of the machine must exist in the real world where these simple conceptual models
FIGURE 3.1A

FIGURE 3.1B

CELL-BUS INTERCONNECTIONS
are inadequate.

Later in the chapter, a realization of a "bus" with even more magical properties will be developed. This bus-like structure will propagate signals in either direction selectively such that left and right going signals never reach a cell site at the same time.

An APL program (Appendix C) was written to simulate the machine described herein. The illustrations provided are taken from that simulation.

The cell functions are very similar to those described in Table 1.4. Each cell has the capabilities: (1) of examining the bus (at a "window" in the region of the cell and changing states according to the bus command; (2) of transmitting to the bus in a vacant window; (3) or of directing the cell to its right.

The branching operation as in the clocked version requests an associative match for a particular label and then transfers control to that label. Spatial separation on the bus is used to order the constituents of a label. The bus is terminated at each end by its characteristic impedance so that signals reaching the end are absorbed. The process of locating strings in the *set-to* sequence is equivalent to the search for
branch names. The asynchronous copy operation differs dramatically from the clocked copy operation.

Recall that in the clocked machine the copy string was directed character by character until the string was exhausted. This is a reasonable approach when the amount of time required to retrieve a single character is a small constant. When the time required to access a character varies with the position of the string it is reasonable to attempt to find a burst mode of character transmission. This is accomplished by adding a command to the bus repertoire which directs a located string to transfer itself to the bus. The difficult thing with this burst transfer mechanism is to decide when the transfer is complete. The time required to make this decision is related to the size of the machine and the relative locations of the accessed string and the directing string. Another single rail bus with the same transmission co-efficients as the symbol bus is provided to sense the "size" of the machine. By placing a positive pulse on the bus and terminating the line appropriately the cell may sense completion of the transfer by awaiting the reflected pulse. The bus termination must be chosen in such a way that the pulses are reflected only once. An analog network is shown in Figure 3.2 that reflects the
negative of a positive pulse and absorbs a negative incident pulse. In this way it is only necessary for the cell to count two incident negative pulses. It is not necessary for the control pulses to travel to the extremities of the array. The structure of the machine requires that the copy cell, \( D \) in the examples that follow, be the extreme right cell in use. This means that it is not necessary to search further to the right for a name. Similarly the cell containing the found name is the furthest cell to the left that need be searched. The cells may be designed to conditionally "short" the bus to reflect the positive pulses. Figure 3.2 contains an example of this situation as well.

The following snapshots are successive states of the \( S^2P \) machine. The APL symbols used include: \( \downarrow \) for label; \( \forall \) for set; \( \rightarrow \) for to; and \( \rightarrow \) for goto. The example is a program string that reverses a string named XYZ. No attempt is made to illustrate the if-then mechanism as it does not represent any new ideas beyond the goto and set-to constructs. The term cycle refers to the length of time it takes a bus signal to move between adjacent cell sites. "Cycle" is taken from the familiar analog in the design of classical machines.
FIGURE 3.2
CONTROL BUS TERMINATIONS
The sequence is initiated by forcing the L cell of LABC to the 'Z' state. An examination of the S²P syntax reveals that it is indeed possible for a string to have more than two names. Should a branch in control be made to the first name in the string, the second label will be "in the way" and the control signals must propagate through the label without perturbing the bus.

The first snapshot in Figure 3.3a shows the propagation in progress-- the negative command cell activity denotes this idle condition. An unusual problem arises when the execution of the program requires that one of the two (or more) names for a string be removed. Should the removed name be the leftmost one of the label part then the name can be removed by simple erasure. However if the removed name lies within the label part special care must be taken to remove this name without damaging the string. Garbage collector considerations are treated in detail in Chapter IV. When control reaches the V cell, search for a label begins. The information packet |+1Z|+1Y|+1X|-1L| moving to the right is searching for the name and a similar packet is moving to the left. The differences in spacing of the left and right information trains are due to the Doppler
shift. The source of bus signals, the command cell, moves to the right so that the frequency of symbols propagating from the right must necessarily be slower. This separation of symbols is useful in later operations and some attempt is made to preserve cell separation for cells originating from a command cell that does not move. This is a consequence of the assumed nature of the bus and is not a necessary condition in other configurations.

The + (to) cell must direct two similar operations on the bus. The string that is to be replaced and a new string with the same name must both be tagged. This is done by the +0/:p and +0/; cells respectively.

Once the -0/: directive has been placed on the bus the command string can begin expression evaluation. In this case the search is for the string with the same name. The search for the original set name has progressed sufficiently by Snapshot 8. The +1/: marks the search string; by the next frame (Snapshot 9) the string has been tagged for ultimate deletion.

Before this matching operation has taken place the command string has provided signals to find the replacement string and to locate its tail. The frequency change (separation between the +2: and +2u
bus directives) is due to the above-mentioned desire to maintain some separation in left going commands. The necessity of this approach will become clear when the instructions reach the object string. Even before the set string has been located the machine is paused, waiting the copy of the tail of XYZ as indicated by the +0/*p cell as seen in Snapshots 8 and following. The graphic symbols 1 and τ are used to represent positive and negative pulses on the control bus. The -0* directive will initiate the required copy.

Meanwhile on the right side of the array, the new name LXYZ is being copied as is shown in Snapshots 8 and 9. The 'M' state is used to inhibit the copy function for one cycle to avoid copying the bus contents which have already been copied into the cell on the left. This lock-out technique with the bus caused by the cell change directives is necessary to preserve timing relationships. This relationship may seem awkward in the bus configuration as described but when the cell delay dominates as in the realization presented later such an interlock is easily achieved. The locking signal does not occur for signals moving to the left as it is unnecessary and will cause difficulties in the garbage collector. The cell then must have the power not only to sense bus activity but the velocity of such
a disturbance as well.

By Snapshot 10, some 37 command cycles into the execution of this string, the new name \( \text{name} \) has been formed at the right and the locating of \( \text{XYZ} \) is in progress at the left.

Two frames later the -0* directive has enabled the copy cell to copy 2 active bus symbols (the copy cell is in the 'P' state). Notice also the negative reflection on the control bus caused by the copy cell. By this time the tail operation has marked the +2/Q cell to eventually begin the copy operation. The "bouncing" of the 1 on the found string is not implemented in this version of the simulator.

The 'C' state is shown directing the placement of the to-be-copied symbols on the bus. The operation terminates on the #. In the final state configuration of Figure 3.3d the array is quiescent; the +0/*p will next become active after receiving the two \( \tau \) pulses. Notice the count down to state 'Q' in the following frame. Only the information packet ;SRQ remains on the bus with the \( \tau \) pulse almost incident at the +0*Q cell by Snapshot 19. Some time later the ;SRQ packet has been copied and the search for the head of XYZ is in progress. This condition persists without complication until the head operation
takes place at the left of the array in Snapshots 29 and 30. First the head of XYZ is marked by the \( +2/.p \) activity state. This permits two things to happen. If the program request is of the form of a single head followed by a copy request then the cell will place only its contents on the bus and revert to its former quiescent state. Should an additional request such as tail follow the head request then the marked cell will return immediately to the idle state as the tail of a single character is null.

Having placed the \( +2P \) on the bus, the array waits for the copy operation to take place. The ; cell continues to count negative control-bus pulses. The process of searching for a new name begins in Snapshot 40. The name is copied while the goto proceeds and is returned to the array for garbage collection by terminating it with a \( \neq \). The \( +0:/P \) cell that has waited patiently for annihilation is finally satisfied by receipt of the \( -0\neq \). The new string XYZ was formally inaugurated when the \( \div \) cell received the \( -0\neq \) bus directive between Snapshots 41 and 42. The program forces a transfer to ABC and would remain in this loop indefinitely. Three garbage strings are created in each pass through the loop and are consumed by the garbage collector (see Chapter IV).
STATE

B
U
S

C

E

L\{XYZ:PQRS;=[ABC:VXYZ+XYZ+XYZ<;\rightarrow ABC;=]\}=

12 SNAP 4

SHAPSHOT=1 CYCLE=1

B
U
S

C

E

L\{XYZ:PQRS;=[ABC:VXYZ+XYZ+XYZ<;\rightarrow ABC;=]\}=

SHAPSHOT=2 CYCLE=5

B
U
S

C

E

L\{XYZ:PQRS;=[ABC:VXYZ+XYZ+XYZ<;\rightarrow ABC;=]\}=

SHAPSHOT=3 CYCLE=9

B
U
S

C

E

L\{XYZ:PQRS;=[ABC:VXYZ+XYZ+XYZ<;\rightarrow ABC;=]\}=

FIGURE 3.3 A
SNAPSHOT = 8  CYCLE = 29
B  1 -  -  +  +  +  -  -  +  +++  -  +  +  =  -  +  -  +  ++  -  -  -
U 00 02 2 2 2 2 2 2 0 02 2222201111
S  X  Y  Z  :  u  *  *u  :  ZYL  :  ZYL

SNAPSHOT = 9  CYCLE = 33
B  -  +  +  +  +  -  -  +  +++  -  +  +  =  -  +  -  +  ++  -  -  -
U 02 2 2 2 2 2 2 0 02 2222201111
S  X  Y  Z  :  u  *  *u  :  ZYL  :  ZYL

SNAPSHOT = 10  CYCLE = 37
B  +  +  +  +  +  -  -  +  +++  -  +  +  =  -  +  -  +  ++  -  -  -
U 2 2 2 2 2 2 2 0 02 2222201111
S  X  Y  Z  :  u  *  *u  :  ZYL  :  ZYL

SNAPSHOT = 11  CYCLE = 41
B  +  +  +  +  +  -  -  +  +++  -  +  +  =  -  +  -  +  ++  -  -  -
U 2 2 2 2 0 02 2222201111
S  Y  Z  :  u  *  *u  :  ZYL  :  ZYL
16 SNAP 4
SNAPSHOT=20 CYCLE=77
B
U
S
C
E
L
Q
P

SNAPSHOT=21 CYCLE=81
B
U
S
C
E
L
Q
P

SNAPSHOT=22 CYCLE=85
B
U
S
C
E
L
Q
P

SNAPSHOT=23 CYCLE=89
B
U
S
C
E
L
Q
P
The need for macro facilities will be discovered as soon as one attempts any reasonable programming task in $S^2P$. With the structure as now developed a simple macro process will be pursued. Its limitations will necessitate the development of a more sophisticated macro arrangement although the improvement is not possible with the hardware heretofore described.

The simple macro facility uses a very restricted copy rule. For a macro defined:

```
\text{MAC:XXX,YYY,ZZZ#}
```

the macro call [MAC,AAA,BBB,CCC] will result in the execution of the program string:

```
XXXAAAAYYYBBBBZZZCCC.
```

Simple substitution is all that is permitted; no nesting of macros is allowed. The implementation process is obvious enough. "[" is a request for a macro call that is initiated by using a \text{-ll} directive. When the "," after the name is reached, control is passed to the searched string and the "," enters a waiting state; each time a comma is encountered a branch back to the preceding comma is made. The \# terminating the end of the macro also passes control to the tagged comma, but the macro cannot be accessed further. This does not require a more complicated \# cell function as the effect of \#
may be modified to always permit passage of control to the sending (calling) comma cell whether it exists or not. The ']' which terminates a macro call also cancels any sending comma forms within the macro if they exist. It is, therefore, permissible to have fewer arguments to a macro call than are necessary; but more arguments than the macro can support leads to the unfortunate situation of passing control to a non-existant location.

A subtlety of this approach is that at each comma-transfer request the current cell activity must be passed. This is achieved by using the activity/! configuration to link between segments of the macro and the cell, forcing the cells to disregard matching requests when the bus symbol is a !. That is, only comma cells interpret the activity of activity/! directives. The other cells treat this command as +0/∅. Such a macro facility will permit easy table look-up operations and repetitive substitution operations as shown in Figure 3.4. The macro HAND provides the framework in which to substitute the cards of the various suits. By this point the concept of a transmission line like bus has served its usefulness.

The next section describes an asynchronous network
labelH: HAND, SPADES, DIAMONDS, CLUBS/
[HAND, EAST, KJ106, AQ432, J53, A]
[HAND, SOUTH, Q975, 56, AK10984, 3]

Expanded Macros

HAND EAST
SPADES KJ106
HEARTS AQ432
DIAMONDS J53
CLUBS A

HAND SOUTH
SPADES Q975
HEARTS 56
DIAMONDS AK10984
CLUBS 3

FIGURE 3.4
Simple Substitution Macros
that has the necessary properties required for an iteratively structured information processor.

An asynchronous network is desired to simulate the actions of a transmission line as required in the development of the $S^2P$ machine. The design goals include:

(1) Facilities for the propagation of information to the left or to the right.

(2) At any cell site only one directive may be handled at a time; in a situation where signal flow is both left and right preference is given to those signals moving to the left although every directive reaches every cell on an interleaved basis.

(3) The order of the symbols on the bus is preserved; the spacing between symbols is irrelevant.

(4) Cell directives and bus directives remain in lock step.

(5) Direction of motion is preserved and such information is available to the cells.

(6) It is possible to selectively terminate the bus at any cell site. Selection features include-- but are not limited to-- bus contents, cell contents, and direction of motion.

The understanding of how such a structure is developed is provided by the unilateral bus of Figure 3.5.
FIGURE 3.5
UNILATERAL ASYNCHRONOUS BUS NETWORK
Extensive use is made of the ready-acknowledge form of commands. A command produces a ready signal when it desires to initiate a sequence. The completion signal then responds with an acknowledge. The double bussed RDY-ACK signal uses a form of transition logic[3.1]. If the signals are different, then the line is active; and if they are the same, the idle condition prevails. The generation of RDY merely changes the level of one of the lines while the generation of acknowledge changes the other. The XOR function will then sense the appearance of a ready signal. In the unilateral bus example the generation of a ready signal will clock data from the left into the new cell provided that the old data has been transferred to the right. Once the data has been transferred, the acknowledge signal is sent back to the left. When the cell has completed its operation, it releases the bus for further propagation by generating a ready signal on the right.

The development of the bus network must pay proper attention to the nature of the asynchronous control circuitry. All unnecessary delays should be eliminated and the dependancy on transition time minimized. The bus register will consist of a number of bits sufficient to include the width of the bus and some error checking circuitry. Since signals may be
propagated in both directions, it will be necessary to load the bus from two sources. It is assumed that the data propagation rate equals the control signal rate so that by the time the actuating ready signal is received, data must be available at the register. This means that there can be no gate delays between the data and register to switch the left and right propagation signal. A double D flip-flop design is shown in Figure 3.6 that can be loaded from two different data paths by two separate clocks. The restriction placed on the clock lines is that at most one clock be on at any time; data transfer is on the leading edge of the clock line-- this rising edge trigger operation is assumed for each variety of flip-flop in the design. The set and reset lines are normally high. At most only one of \( R, S, C_1, C_2 \) may be high for the flip-flop states to be deterministic. The use of this flip-flop is illustrated in Figure 3.7. The RDAT and LDAT lines are data from the right and left cells respectively; RXFR and LXFR are clocking signals to the flip-flop. BXFR is a transfer signal to a hidden register that is used to allow the crossing of data streams from the left and right. The generation of \( R,L,BXFR \) is described in the next paragraph. BDAT and SETBUS are
FIGURE 3.6
DOUBLE D FLIP-FLOP
FIGURE 3.7  Bus Register for an ISIP
used within the cells to write data on the bus.

Figure 3.8 illustrates a simplified version of the bus control network. The inclusion of the bus-cell state directive interlock has been omitted for clarity; the cell state directive is transferred in the same manner and inhibits certain acknowledge signals depending on the cell contents-- such a design is straightforward and is not included here. The signals DONE, PROPL, and PROPR are also cell content dependent and combinatorial; the specifications of such functions is below the scope of this thesis.

The top half of the afore mentioned figure is concerned with the propagation of signals to the left. It is a near mirror image of the circuit of Figure 3.7. Two additional features have been added to handle the bi-directional case:

(1) A cell may not receive data from the right unless it is not busy and

(2) It has transferred its contents to the left--as indicated by the equivalence gates in the lower right corner--or the buffer register to the right has been loaded--as indicated by the BUFFILD signal.

The chain of a pair of ready-acknowledge signals at the upper left permits the interleaving of signals from the left and right. In fact, it forces such a condition if
FIGURE 3.8
BILATERAL ASYNCHRONOUS BUS NETWORK
such is possible so that a long string of closely packed symbols from one direction or the other cannot hang the bus. The remaining RDY-ACK pair is used to transfer data from the left. The delay element is necessary to assure that the hidden buffer register has stabilized before the bus register is clocked from the left. The purist can be satisfied by using an additional RDY-ACK network in place of the delay. The internal signal is cleared on any clock of the bus register and set when the symbol has been interpreted. LXFR, RXFR also set internal motion flip-flops so that the direction of propagation is maintained. Toggling $Q_2$ and $Q_4$ with PROPL and PROPR appropriately will cause propagation of the symbol; by changing the states of the motion flip-flops the bus may be effectively terminated anywhere throughout its length. An asynchronous network has been presented that satisfies the design criteria.

A quick inspection will reveal that $S^2$ contains only three operators, two of which are unary and the third is binary and associative. As all the dyadic operators are associative, there is no need for parenthesized structures. As the number of operators is increased, the desirability of changing the implied associativity of the operators increases. An obviously
useful extension to the operators in $S^2P$ would be the Boolean operators NOT, AND, and OR for use in the IF-THEN conditional clause. This operator set is, of course, not completely associative. A first-in last-out storage structure, a stack, is often used on conventional machines to implement these facilities. If the iteratively structured information processor is to be valuable, it must certainly include a stacking mechanism.

The property of the bus that permits propagation selectively in either direction makes it possible to realize a stack. Recall that the strings are built only at the right edge of the array. If the array is arranged in a circular fashion as in Figure 3.9 and information is placed on the bus so that it propagates only to the left, then the most recently created string will be the first one accessed. Sing special tokens to mark the bus--to chop the bus into sections and keep the stack entries from interacting--expressions of arbitrary complexity may be realized. The stacking mechanism need not be restricted to temporary results in string expressions. A block structure may be realized by using BEGIN END brackets to control the stacking process. The bilateral search is faster than the unilateral one and should be used in the outer
block. The statement LOCAL <name> will put the name on the top of the stack. The left search algorithm insures that the most recently formed name will be the first found. Once the string has been located, a cancel signal is sent into the stack so that only one named string is marked.

The ordering of the bus makes it possible to make some significant improvement in the macro facility. Nested macro calls are now possible because the current macro and its call are adjacent in the array. The passage of control between comma's must switch from left to right depending on the location in the call or the macro body. The decision about the direction to pass control is enhanced by introducing the symbol ~ (meaning argument) so that 's searches for the first sending string to the left and ~ passes control to the first sending string to the right. Figure 3.10 illustrates the proposed macro facility. Now the ability to modify bus contents selectively comes into play. If an argument number is permitted behind the ~, a bus directive is introduced that can count commas as it passes them within an active macro space and pass control to the comma that causes the bus contents to go to zero; a form of GPM is thus realized that is deficient only in its ability to handle
FIGURE 3cJO  ENHANCED MACRO FACILITY
recursive macro calls. The discussion of recursive and parallel processes is deferred to the next chapter.

Additional functions may be added to improve the efficiency of $S^2P$.

A dramatic improvement may be obtained by the introduction of the trunk operator. Quite often in string processing it is desirable to remove the first character in a string and operate on it. This is currently handled in $S^2P$ by the following code:

```
set TEMP to XYZhead;
set XYZ to XYZtail;
```

which must locate and copy string XYZ although it is clear that XYZtail is shorter than XYZ and could be retained in the same space. The trunk operator returns a value of the head of the string and has a side effect of shortening the string by one character.

This makes the problem of counting loop iterations quite simple by using unary arithmetic—e.g.

```
set COUNT to '111111';
LOOP: if COUNTtrunk='' then goto FIN;
```

'body of loop'
The trunk operator marks the string as the head operator does, but after placing the lead character on the bus, the cell contents change to the $\phi$-- which is the symbol used in the compactor phase of the garbage collector. A very minor sophistication of the hardware has led to a major software improvement.

Having already permitted an arithmetic facility for indexing arguments in macro calls, it seems reasonable to provide some more powerful arithmetic operations. The relational may be used to provide a sorting feature using the if-then construct; its similarity to the = relational is obvious. By using a coding system where the character bit patterns are numerically ordered in the same way that the characters are lexically ordered-- such as ASCII-- the cell function comparison may use a simple adder. This same adder circuit may be employed to do bus matching so that a separate parity circuit is not required; cell complexity has not increased dramatically.

As soon as the adder becomes a part of the cell, it is reasonable to assume that a state directive which contains carry information may be introduced. Decimal
summation can thus be performed by storing the digit representation of the numbers in inverse order and streaming one addend past another.

There is a very real possibility of streaming a constant across an entire vector-- the symbols pass over the entire vector on the bus anyway so this fortunate happenstance should be exploited. This is the familiar APL situation where \( pX \) in \( X \oplus Y \) may be arbitrary and \( pY \) is null, that is, \( Y \) is a scalar.

There are indications that iteratively structured information processors may be useful in handling iterative structured languages such as APL. APL's basic data structure is a rectangular array of arbitrary dimension. The implementations of APL on conventional machines store these arrays in row major order. That is, the structure is mapped from its \( N \) dimensional space to a linear one which is exactly the environment of the iterative machine that has been discussed. Special punctuation to describe structures has been used to enhance string data-- see NEUCEOL[2.3]. Appendix D provides an example of LISP written in GPM-- the macro facility that has been suggested for \( S^2P \).

The possibilities for interactive realization of complex languages seems limited only by the design
engineers' imagination.
Reference for Chapter III

Chapter IV

Theoretical Bases for Iterative Language Machines

Having established the feasibility of iterative language machines in the previous chapters the remainder of this dissertation will serve to provide some theoretical limitations on such devices.

The syntax for almost all existing programming languages is restricted to a simple string structure. This limitation is deeply rooted in the hardware input/output devices used in human communication with computing machinery. The desirability of this representation is not supported in this discussion. Likewise the extended power of two-dimensional languages will not be pursued. This thesis is rather directed at the iterative realization of language machines having a linear syntax. This is not to say that multidimensional arrays are not useful. In a later section of this chapter, several uses for two dimensional structures are advanced although the basic array is linear. A working definition of an iteratively structured information processor is
provided and a collection of theorems based on this structure is then developed.

A linear cellular language machine (LCLM) is an iteratively structured information processor consisting of a linear array of identical cells each of which communicates with its nearest neighbor. The distinguishing thing about the array is its ability to execute a higher-level programming language directly. The ability to scatter information from one cell throughout the array is accomplished by chaining the information path through each cell. This chain is given the name bus to distinguish its operation from other forms of intercellular communication such as state transition information between adjacent pairs of cells. The unique properties of this bus include:

1. the selective propagation of information in either direction along the bus;
2. left and right going signals on the bus pass each other without interference and never make simultaneous demands on a cell processor;
3. the ordering of information on the bus is preserved but the spacing is irrelevant.

Theorem 0--A LCLM with the required structure exists.

The proof is the example of the preceeding
An important feature of the machines described here is their ability to achieve efficient storage management by the use of a companion garbage collector that locates and removes unwanted program segments and reorders the remaining segments. The garbage in the processor is of three types: (A) unnamed strings created by the operation of the machine itself; (B) unreachable named strings that cannot be used by the operating program because code containing the references may not be executed; and (C) named strings with no explicit references in the program.

Theorem 1-- The dynamic garbage collector in an LCLM operates without interference with the process-active portion provided the program string does not delete or modify itself.

Proof: Take any cell in the array. During the isolation process the garbage collector and the program processor may be in conflict if a cell is marked for deletion and an applicable bus signal is present or a cell state directive is incident on the cell in question. The latter condition is in violation of the hypothesis so this conflict of interest is avoided. The first condition does not represent a conflict; either event, deletion or direction, may take place
because even though the bus action is handled by one cell, that cell is in a garbage string and will be handled by the garbage collector anyway.

In the compaction phase an examination of the compactor cell $\phi$ and its nearest neighbors by case will complete the proof of the theorem. In the absence of bus directives the nature of the $\phi$ cell in handling inter-cellular communications assures that no interference occurs. Recall from Chapter I that the $\phi$ passes signals unchanged. Knowledge of the direction of bus disturbances (an assumed property) leads to a garbage collector design without conflicts.

Case I: Bus Signals from the Left
(See Figure 4.1.) Bus directive A forces an intercell communication C. The property of the bus, intercell interlock, dictates that A and C reach cell $\phi$ at the same time. The function of $\phi$ cells on interlocked signals is of course not to release the locking mechanism so that A-C arrive at y in synchronism.

Case II: Bus Signals from the Right
(See Figure 4.2). If the signal at B does not exist no problems occur. For closely spaced signals the $\phi$ symbol must direct traffic to separate the closely packed directives. The $\phi$ cell with a C directive will then arrive at the y site as required. Therefore
FIGURE 4.1

Garbage Collector with Bus Disturbance from the Left

FIGURE 4.2

Garbage Collector with Bus Disturbance from the Right
the class A garbage problem is solved for a linear cellular language machine.

Theorem II. There is no way to identify class B and C garbage and thus no way to deal with it.

The proof is analogous to the classic halting problem. Replace all locations where a reference to a named string occurs by a branch to a non-existent string, the halting configuration of an iteratively structured information processor. There is no way to decide if the halting state can be reached and therefore no way to identify class B garbage. The class C problem is a special case of B in which a reference must first be produced.

The action of the garbage collector in $S^2P$ is indicated in Figure 4.3. The isolation phase quickly locates the dummy label created by the branch instruction and the compactor phase is initiated almost as rapidly. As with the control bus termination problem the propagation of $\Phi$ cells past the copy cell is unnecessary; this fact is not exploited in the simulator.

A very useful property of iteratively structured machines is the ability to support more than one process. It is a simple matter of examining recursive and parallel process definitions to show that these are
STATE

\[ \begin{array}{cccc}
B & - & + & + & - \\
U & 0 & 1 & 1 & 1 & 0 \\
S & u & A & B & C & u \\
\end{array} \]

\[ \begin{array}{cccc}
& + & + & + & + \end{array} \]  

\[ \begin{array}{cccc}
L & X & Y & Z & P & Q & R & S \\
\end{array} \]

\[ \begin{array}{cccc}
M & uC & B & A & u \end{array} \]

\[ \begin{array}{cccc}
P & \phi & \phi & \phi & \phi \end{array} \]

3 SNAP 1

SNAPSHOT=44  CYCLE=179

\[ \begin{array}{cccc}
B & - & + & + & - \\
U & 0 & 1 & 1 & 1 & 0 \\
S & u & A & B & C & u \\
\end{array} \]

\[ \begin{array}{cccc}
& + & + & + & + \end{array} \]  

\[ \begin{array}{cccc}
L & X & Y & Z & P & Q & R & S \\
\end{array} \]

\[ \begin{array}{cccc}
M & uC & B & A & u \end{array} \]

\[ \begin{array}{cccc}
P & \phi & \phi & \phi & \phi \end{array} \]

SNAPSHOT=45  CYCLE=180

\[ \begin{array}{cccc}
B & - & + & + & - \\
U & 0 & 1 & 1 & 1 & 0 \\
S & u & A & B & C & u \\
\end{array} \]

\[ \begin{array}{cccc}
& + & + & + & + \end{array} \]  

\[ \begin{array}{cccc}
L & X & Y & Z & P & Q & R & S \\
\end{array} \]

\[ \begin{array}{cccc}
M & uC & B & A & u \end{array} \]

\[ \begin{array}{cccc}
P & \phi & \phi & \phi & \phi \end{array} \]

SNAPSHOT=46  CYCLE=181

\[ \begin{array}{cccc}
B & - & + & + & - \\
U & 0 & 1 & 1 & 1 & 0 \\
S & u & A & B & C & u \\
\end{array} \]

\[ \begin{array}{cccc}
& + & + & + & + \end{array} \]  

\[ \begin{array}{cccc}
L & X & Y & Z & P & Q & R & S \\
\end{array} \]

\[ \begin{array}{cccc}
M & uC & B & A & u \end{array} \]

\[ \begin{array}{cccc}
P & \phi & \phi & \phi & \phi \end{array} \]

FIGURE 4.3 A
instances of the same problem and hence may be solved by the same mechanism. A recursive task involves the request for the existing task space for separate purposes; the parallel process involves two tasks requesting the same space at the same time.

Dijkstra[4.1] has demonstrated the validity of the stacking process to handle recursive programs. Several methods of achieving this stacking are available. The conflict cell may pause and await the appearance of an empty cell $\emptyset$ and use it for a copy of the conflict cell tagging the copies appropriately. Such an approach is possible because the conflicts occur only on single cells and are often of very short duration. An embellishment to this technique would require that the conflict cell take steps to assure the rapid generation of the needed empty cell.

A more elegant stacking operation occurs by adding a second dimension to the array creating a quadratic cellular language machine or doubly iterative information processor. The conflict problem is resolved by immediately producing a copy in the dimension orthogonal to the control string. The additional uses of the other dimension are manifold.

Some comments about edge effects seem to be in order. The bounding of the quadratic array with
absorbing terminations as was done with the simple 
$S^2P$ machine is an obvious choice but the interesting 
effects achieved by the circular arrangement of the 
LCLM would lead directly to a toroidal arrangement of 
the surface. Possible array topologies are presented 
in Figure 4.4.

An especially nice arrangement of the quadratic 
array would be to form a cylinder with principal axis 
parallel to the string length and forming a torus-shape 
by twisting a single row at the join of the two bases. 
This gives all the nice properties of the torus and 
also restricts the array to a linear ring in the 
dimension of the program string. This should be very 
beneficial in multi-process environments. 
Short-cutting transit time through the unused portions 
of the array using diagonal transmission modes should 
be investigated.

The identification of the several processes may be 
made by assigning a unique process number to each 
process. This process number is retained by the cells 
and also transmitted as part of the bus symbol so that 
a necessary condition for cell operation is a process 
match. The design of a single process did not require 
a bus structure where the "doubling" of information at 
the cell site could be avoided because the single
Figure 4.4: Array Topology

(A) Linear

(B) Planar

(C) Cylindrical
process puts information on the bus so that information flows in only one direction toward any point. The additional design constraint makes multiprocessing possible.

The task of assigning unique process numbers can be accomplished in a manner analogous to embedding tree structures in conventional memories[4.2]. A process that invokes a parallel process replaces its process number by \(2N\) and initiates the concurrent process with a number \(2N+1\). Inspection reveals that in a binary representation these operations are a simple left shift and bit set operation. Resynchronizing process numbers involves a scale right shift. To maintain uniformity in macro or procedure calls, the call is given a process number \(2N+1\) and the call point retains an activity \(2N\). Parameters covered by the process number are of course global to the active process. The process number 0 does not fit into this scheme as it generates itself and could possibly start process with equivalent numbers. A very reasonable use for process zero is a system maintenance function that performs dynamic fault checking of the array. Such an approach leads to graceful degradation in system performance.

**Theorem III.** The execution of an arbitrary number of processes in an iteratively structured information
processor is possible.

The proof is by induction, the basis step having been the demonstration in Chapter III. Figure 4.5 will prove useful. The non-interference nature of the bus implies that more than one process may exist. Assume that the array supports N-1 processes without conflict. Then in an array with N active processes select the process with the highest process number. (There are a finite number so this upper bound must exist.) Only the cells with process number N or those covered by N when in a procedure call will interact with the bus. Lump these cells of process N into another process and call it a and place everything else in another process b. The theorem holds for N-1 processes so it holds for the process pair a,b and therefore the iteratively structured information processor can support parallel processing. The concept of initiating parallel processes and determining the suitability for parallel processing is not germane to this thesis.

At this point it will be noted that the mechanism for handling arithmetic operations is valid for addition and subtraction but will not work for multiplication as the process is essentially a serial one on a finite state machine and a serial finite state multiplier does not exist. The overflow problem (carry
into a cell not part of the operating string) in the iteratively structured information processor requires the acquisition of an additional cell which can be accomplished by attaching a $\emptyset$ cell much in the manner suggest in the recursive function handler.

The limitations imposed on the language by the iterative technique are concerned with the grammar and the ultimate size of the machine.

Theorem IV. A necessary condition for efficient execution of a language in a linear cellular language machine is that the grammar generating the language be of type LR(1).

For a description of LR grammars the reader is refered to Knuth[4.3]. The proof is by contradiction. The information available to a cell is in the form of an activity and a state. The current cell content is the input token; the basic iteratively structured information processor is therefore using a single character look-ahead. Assume that another character is required to produce the correct bus directive. That is, the grammar is of type LR(2). There is no guarantee that the next character is in an adjacent cell so that the information must be provided by additional connecting links from the surrounding cells. But there can be no additional links to cells other
than the nearest neighbors of the cell in question so that LR(2) grammars cannot be supported in the linear cellular language machine.

The size of the cells of an iterative language machine depends on the semantics of the language employed. Determination of the number of bits of storage required for a particular semantic variation is provided; the extension to other semantic forms is obvious. In a language where each operation is applicable to only one type of data (for instance, $+$ is an integer ADD request and $+$ is a logical OR operator instead of using the same $+$ symbol differently in context) then the width of the cells may be determined by the following formulas:

(1) The number of activity bits is $1 + \sum \log_2$ (number of statement types). The sign of the activity directs conditional copy modes and the activity bits are used to locate constituents required by the various statement types.

(2) The number of symbol bits is $\sum \log_2$ (number of symbols in the alphabet +3). The three additional symbols, □, $\phi$, $\omega$ are used in garbage collection and copy operations independent of the language.

(3) The number of states required by the machine
is bounded above by twice the number of special symbols. The restriction to mono-typical operators dictates that only a single send-receive pair for each operator will be needed.

Using these estimates as guide lines, the cell size of the $S^2P$ machine is thirteen bits, six for symbol storage and seven for activity state information or approximately fifty percent of the memory of the array is to be devoted to the operating system. This ratio is typical of existing mini-computer operating system-user space ratios. The use of a portion of memory to distinguish operand types is a popular idea in current machine architecture[4.4]. The distribution of the control circuitry throughout the memory (array) is an extension of this tagging technique. The ultimate size of the iterative machine is one of economics and not design limitations, due to its periodic architecture.
References for Chapter IV


Chapter V

Summary, Conclusions and Extensions for Machines of the Iterative Architectural Style

A novel machine architecture has been developed based on the concept of the direct execution of higher-level programming languages. The structure of the languages making efficient use of this architecture has been formulated. Techniques for establishing parallel processing have been demonstrated and the rudiments of a programming language to exploit these features have been mentioned. The resulting machine design seems to be well-suited for several classes of computer systems problems -- those dealing with one-time problem solving sessions where the desk calculator approach is impractical and those problems related to massive collections of similar processing that are non-serial in nature. In addition to the obvious speed improvement, the support of multiple concurrent processes means that a dynamic fault analysis procedure may exist during execution to permit fault detection and graceful degradation of system performance.
This thesis provides a non-trivial contribution to the state of the art by demonstrating another iterative arrangement for a computing machine. This architecture differs significantly from other iterative architectures in that it is far easier to program because the iteratively structured information processor executes a higher-level programming language directly. The design represents an innovation in the area of machines designed for direct execution of higher-level languages in that it makes exclusive use of an associative memory which eliminates any pre-pass (symbol table building) phase. Although the machine is particularly well-suited to string manipulation languages, it can be used to realize any language whose structure can be mapped onto a string. By far the most important feature of the machines described in this thesis is the ability to support parallel processing. The streaming of data along the bus and the possibility of concurrent instruction sequences together with the iterative architecture which allows construction practice to cater to speed requirements indicate a very rapid machine. A significant improvement in parallel process machine design is made possible because the machine is executing a higher-level language; the programmer's description of a process coincides with
the machine execution.

There are several extensions of this design that should be pursued. An attempt at hardware realization of a simple language is a first logical step. A SNOBOL-like pre-processor in this architecture is a possible candidate for such implementaion. Construction techniques for such machines represent an area of research. The design of peripheral equipment for these machines must be investigated.

The availability of a machine that effectively executes parallel processes suggests the development of languages to exploit this property. The use of multidimensional arrays has not been exploited completely. The possibility of using a multi-layer architecture in which each layer executes a different programming language does exist. The specification of the individual layers, their geometries and the interaction between layers offers a multitude of research problems.

It is the author's belief that research in computer engineering must be carried out in advance of the state of the art with a cautious respect for the direction that implementation technology is progressing. Toward this end, parallel processing techniques and array structures must be considered.
Caxton Foster [5.1] has projected a list of principles to be found in machine architecture by the year 2000. These include:

- an interpretive engine capable of executing directly one or more higher level languages,...
- a small wired-in operating system,...
- [the machine is to] be privately owned and monoprogrammed.

On these points, iteratively structured information processor design is in exact agreement. The enhancement of the utility of computer systems is, as indeed it should be, the goal of all computer engineers/scientists. The continued development of Von Neumann based machine architectures and the software support for them in the form of compilers and single process operating systems may be misdirected. The need for intelligent processors with minimal software support is not in the distant future, it is in the present.
Reference for Chapter V

Acknowledgement

The patience and assistance of Dr. Robert Minnick, who served as thesis director, will long be remembered. The consideration of the other thesis committee members, Dr. Sigsby Rusk and Dr. Kenneth Kennedy, is also appreciated. The following persons were particularly helpful in the production of the manuscript: my wife, Cathy, and Messrs. Stan Wileman, Peter Meigs, and Randy Neff. This work was supported in part by a NASA Traineeship.
Appendix A

An Annotated Bibliography of Material Related to
the Design of Iteratively Structured Language Machines

Some commonly used abbreviations

From the Association for Computing Machinery:
CACM Communications
JACM Journal
PACM Proceedings

From the Institute of Electrical and
Electronic Engineers:
IEEETC
IEEETEC
IRETEC Transactions on Computers
IEEETSSC Transactions on Solid State Circuits

Computer Conference Proceedings
EJCC Eastern Joint
FJCC Fall Joint
SJCC Spring Joint

Barnes, G.H. et al, "The Illiac IV Computer,"
IEEETC XVIII-6 AUG '68. Array processor
well-suited to problems of a spatially
iterative nature such as partial
differential equations.

Design of a FORTRAN Machine,"
IEEETEC XVI-4 AUG '67. Pre-pass to build
symbol table then direct execution of
FORTRAN text. Drawback is the inflexibility
of chosen language.

Berkling, K.J., "A Computing Machine Based on Tree
Structures," IEEETC XX-4 APR '71.
Conventional memory with unique addressing
mechanism to embed binary trees. Wastes space in memory badly.


Campeau, J.O., "The Block Oriented Computer," IEEETC XVIII-6 AUG '69. Isolation of good process elements—blocks to construct array processor.

Cannon, Lynn E. "A Cellular Computer to Implement the Kalman Filter Algorithm," Tech Report, Montana State University, Department of Electrical Engineering, Bozeman, Montana JUL '69. Array structure added to CPU to do extensive matrix calculations.


arbitrary number of arithmetic machines may exist. Still cannot be programmed.

Crespi-Reghizzi, S. and Morpurgo, R., "A Language for Treating Graphs," CACM XIII-5 MAY '70. GEA Graph Extended ALGOL. Pre-pass phase added to ALGOL to yield graph process extensions. Graphs are treated as dual lists.


Foster, J.M., List Processing, American Elsevier (1967). Elementary description of list processing and an example of the typical language.


Gonzales, Rudolpho, "A Multilayer Iterative Circuit Computer," IEEETEC XII-6 DEC '63. Three layer sandwich machine in the Holland style that is suited to processing visual data.


Holland, J.H., "A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously," EJCC Proceedings '59. A very impractical machine (cannot be programmed). Array of processors that serve as links to data or operate on data input from such paths. Path building and processing can take place concurrently.

Hopcroft, John E., and Ullman, J.D., Formal Languages and Their Relation to Automata, Addison-Wesley, Reading, Massachusetts (1969). Properties of languages and the automata to recognize them. LR(k) grammars and stack automata.

Huang, J.C., "A Universal Cellular Array," IEEETC XX-3 MAR '71. Use of autonomous sequential machine to realize large numbers of functions in a single array. Cutpoint locations are restricted to the boundary.


Kane, Gerald R., "A Useful Cellular Logic-in-Memory Array with a Simple Cell Configuration," unpublished memorandum, Rice University, JAN '72. Modification of the Kautz array that reduces cell Complexity.

Kautz, W.H., "Cellular Logic-in-Memory Arrays," IEEETC XVIII-6 AUG '69. Sorting arrays to build many devices such as cams, stacks etc.


Knuth, D.E., "On the Translation of Languages from Left to Right," Information and Control, VIII (1965). Theoretical remarks on languages that can be parsed looking only
at the stack and the next k characters, i.e. LR(k) grammars. LR(1) is just the right size for interpreted languages.

Lee, C.Y., "Intercommunicating Cells, Basis for a Distributed Logic Computer," FJCC Proceedings '62. Associative memory where cells can interact with their nearest neighbors as well as a common bus.


McCarthy, John et al, LISP 1.5 Programmer's Manual, MIT Press (1959). LISP, the premier list processing language. Very formal language, the model for many related languages.

McCarthy, John, "Recursive Functions of Symbolic Expressions and Their Computation By Machine, Part 1," CACM III APR '60. A description of LISP with emphasis on formalism. Excellent introduction to languages of this type.


Minnick, R.C., "Cobweb Cellular Arrays," FJCC '65 XXVII pt1. Microcellular array with nonplanar interconnection structure which yields arrays that are smaller and more easily repaired than author's cutpoint
arrays.

Minnick, R.C, "Cutpoint Cellular Logic," IEEETEC XIII-6 DEC '64. Description of microcellular structure to realize any Boolean function in array of tailored cells, also contains theory of cellular arrays of this class.


Pratt, Terrence and Friedman, Daniel P., "A Language Extension of Graph Processing and its Formal Semantics," CACM XIV-7 JUL '71. Description of a language to manipulate graphs in terms of its semantics and a realization of the language in LISP.

Reynolds, John C., "Gedanken-A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept," CACM XIII-5 MAY '70. No type declarations—compound structures treated as functions. As name
implies this is a thought language.

Rice, Rex, "Impact of Arrays on Digital Systems," IEEETSSC IV. "For LSI to pay off, a new philosophy is needed for integrating software and hardware using greatly increased quantities of electronics to reduce total costs."


Richards, Martin, "BCPL: A Tool for Compiler Writing and Programming Systems," SJCC Proceedings '69 XXXIV. Clarifies distinction between addresses and values and provides for the manipulation of both. Fundamental item is the bit pattern.

Rosen, Saul, ed., Programming Systems and Languages, McGraw-Hill, St. Louis (1967). Collection of papers dealing with various programming languages. Large section on list processing languages with a very good comparison between SLIP, LISP, COMIT, IPL.

Rux, P.T., "A Glass Delay Line Content-Addressed Memory System," IEEETC XVIII-6 JUN '69. Circulating memory that can be modified-referenced at each pass. Could be the basis for implementing a cellular machine in a non-cellular fashion.


Slotnick, Daniel E. et al, "The SOLOMON Computer," FJCC '62. Array processor with very large cells, all under the direction of a central processor. Forerunner of the Illiac IV.

Design philosophy and construction details of a very sophisticated macro generator originally designed to aid compiler construction. Technique may be applicable in designing macro facility for string processor.

Asynchronous operation of arbitrary sized machine of Sturman's design is achieved by use of transmission line properties within the cell linkage structure.

Linear array of very simple cells based on Lee's associative memory. No compiler or higher-level language is mentioned.


Thurber, K.J. and Myrna, J.W., "System Design of a Cellular APL Computer," IEEETC XIX-4 APR '70. Prepass APL then matrix(32x32) and vector calculations in a cellular arrangement of processors. Limit on vector size is a poor compromise in the design.


Unger, S.H., "A Computer Oriented Toward Spatial Problems," IRE Proceedings XLVI OCT '58. Array of machines with one bit wide accumulators and a few one bit words of memory under the direction of a central
processor.


Weizenbaum, J., "Symetric List Processor," CACM VI-9 SEPT '63. SLIP, a collection of FORTRAN subroutines for list processing. Recursive power added to FORTRAN.


Appendix B

ALP--A List-Processing Language to Embed in ALGOL

ALP is a collection of procedures to provide list-processing capabilities for ALGOL. Its design was motivated by a desire to have efficient list-processing capability in an existing higher-level language. A version of SLIP modified for ALGOL existed at the time ALP was developed and the large number of SLIP subroutines put a large burden on compile time. Additionally, almost all of the list functions in SLIP require some sort of procedure linkage. In ALP the macro facility of Burrough's extended ALGOL is used to reduce such procedure calls. The recursive nature of the host language also means that subroutines to implement the recursive properties need not be provided within the package. The garbage collector mechanism is of the unwanted-list type as in SLIP but the data structure is a binary tree as in LISP.

The ALP package forms the outer block of an ALGOL program. The LISTSPACE array is used to store the lists which are kept as a pair of subscripts of LISTSPACE. These subscripts are called HEAD and TAIL. The sist elements are identified by the use of a flag bit to denote whether the item is a list or an
atom. Atoms occupy two words according to the following recipe. The flag bit of the first word is 0 to indicate that the item is an atom. The head of the first word points to the full word data atom. The tail is null. Full word atoms are formed by the ELEMENT procedure (Figure B.1). This technique does not require the use of a fullword list as in the LISP implementations. LISTS PACKE[0] is always zero; it is the null atom. There are two system lists FREELIST which is a list of all the available words in LISTS PACKE and GARBAGE, the list of unwanted items that may be returned to FREELIST as necessary. The initial configuration of the free list is shown in Figure B.1. Items are placed in GARBAGE by the procedure FREE which tags every item on the primary level of the freed list as garbage. When necessary the garbage collector transfers those items on the GARBAGE list to the FREELIST. Secondary lists (sublists) and shared list management are the responsibility of the programmer. A property sublist may be tagged using the MARK primitive and returned to the available space along with the first level list by the garbage collector.

The ALP package is provided in the following listing.
(A) Structures in ALP

(b) Free List

(c) Garbage

FIGURE B.1 Format of the ALP List System
array LISTSPACE[0:7,0:512];
integer FREELIST,GARBAGE;
label FINIS;
file out PRINTER 1(2,15);
define NULL=0#,
SIZE=4095#,
HEAD(HEAD1)=LISTSPACE[(HEAD1).[14:6],
(HEAD1).[8:9]].[29:15]#,
TAIL(TAIL1)=LISTSPACE[(TAIL1).[14:6],
(TAIL1).[8:9]].[14:15]#,
TAG(TAG1)=boolean(LISTSPACE[(TAG1).[14:6],
(TAG1).[8:9]].[31:1])#,
ATOM(ATOM1)=not(boolean(LISTSPACE[(ATOM1).[14:6],
(ATOM1).[8:9]].[31:1]))#, 
EQUAL(EQUAL1,EQUAL2)=ATOM(EQUAL1) and ATOM(EQUAL2) and LISTSPACE[(HEAD(EQUAL1)).[14:6],
(HEAD(EQUAL1)).[8:9]] =LISTSPACE[(HEAD(EQUAL2)).14:6],(HEAD(EQUAL2)).[8:9]#, 
MARK(MARK1,MARK2)=LISTSPACE[(MARK2).[14:6],
(MARK2).[8:9]].[30:1]:=MARK1#;
procedure GARBAGECOLLECTOR; forward;
integer procedure MAKE(A,B) ;
    value A,B ;
    integer A,B ;
    begin
        integer TOP ;
        while (TOP:=HEAD(FREELIST))=NULL
            do GARBAGECOLLECTOR ;
        FREELIST:=TAIL(frelist) ;
        HEAD(TOP):=A ;
        TAIL(TOP):=B ;
        MARK(1,TOP) ;
        MAKE:=TOP ;
    end of MAKE ;

integer procedure ELEMENT (A) ;
    value A ;
    real A ;
    begin
        integer TOP ;
        while (TOP:=HEAD(FREELIST))=NULL
            do GARBAGECOLLECTOR ;
        FREELIST:=TAIL(FREELIST) ;
        LISTSPACE[TOP.[14:6],TOP.[8:9]]:=A ;
        TOP:= MAKE(TOP,NULL) ;
procedure FREE(A) ;
    value A;
    integer A;
    if a ≠ NULL then
        begin
            integer B;
            GARBAGE:= MAKE(A,GARBAGE) ;
            B:=A ;
            do MARK(1,B) until (B:=TAIL(B))=NULL ;
        end of FREE ;

procedure NEWCELL(A) ;
    value A ;
    integer A ;
    begin
        HEAD(A):=A ;
        TAIL(A):=FREELIST ;
        FREELIST:=A ;
        MARK(0,FREELIST) ;
        LISTSPACE[FREELIST.[14:6], FREELIST.[8:9] .[31:1]]=0 ;
end of NEWCELL ;

procedure GARBAGECOLLECTOR ;
begin
    integer TOP,POPTOP ;
    if GARBAGE=NULL then
        begin
            format FULL("Space Exhausted
                --Terminate Execution") ;
            write (PRINTER,FULL) ;
            go to FINIS ;
        end ;
    POPTOP:=GARBAGE ;
    GARBAGE:= TAIL(GARBAGE) ;
    TOP:= HEAD(POPTOP) ;
    NEWCELL(POPTOP) ;
    while TAG(TOP) or ATOM(TOP) do
        begin
            GARBAGE:=if TAG(TAIL(TOP))
                then MAKE(TAIL(TOP),GARBAGE)
                else GARBAGE ;
            POPTOP:=TOP ;
            TOP:=HEAD(TOP) ;
            if ATOM(POPTOP) then
                begin
NEWCELL(TOP) ;
TOP:=NULL;
end;

NEWCELL(POPTOP) ;
end ;
end GARBAGE COLLECTOR ;

FREELIST:=NULL ;
LISTSPACE[0,0]:=NULL ;
for GARBAGE:=1 step 1 until SIZE do NEWCELL(GARBAGE) ;
GARBAGE:= NULL ;
FINIS: end of ALP.
Appendix C
Asynchronous Operation of a LCLM

The following is a listing of the APL/360 program which simulates the S²P machine. The function APPLY does most of the work with a number of supporting functions. The machine structure is contained in an array C and the character representation of the bus in B.

The action of the transmission line is realized by sliding the two planes of XB with respect to each other using the φ operator. The program flow follows the machine description in Chapter I. The conditional expression feature is not implemented. The interaction of the garbage collector and the intercell state directive is not completely due to the simple structure employed in the rest of the simulation.
\textbf{VLOAD[]}V
\textbf{LOAD}
[1] \textit{T+4} \\
[2] \textit{S+1+2\times p} \\
[3] \textbf{RESET}
\textbf{V}

\textbf{VRESET[]}V
\textbf{RESET}
[1] \textit{C}\leftarrow q(S, 4)p'+0\neq' \\
[2] \textit{C[3;]}'\leftarrow' T,'\square', (1+p)T'p'+' \\
[3] \textit{XB+}(3, S, 2)p0 \\
[4] \textit{XCB+}(S, 2)p0 \\
[5] \textbf{CONVERT} \\
[6] \textbf{CYCLE+SNAPSHOT+0} \\
[7] \textbf{GC+}(2, S)p1 \\
[8] \textbf{GC[;1]}'\leftarrow 0 \textbf{3}'
\textbf{V}

\textbf{VCONVERT[]}V
\textbf{CONVERT}
[1] \textit{B+CH[1++/XB]} \\
[2] \textit{B[;1]}'\leftarrow'BUS' \\
[3] \textit{C[;1]}'\leftarrow'CELL' \\
[5] \textbf{CB[1]}'\leftarrow' \\
\textbf{V}

\textbf{VCHECKPOINT[]}V
\textbf{CHECKPOINT}
[1] \textit{CHXXB+XB} \\
[2] \textit{CHXXCB+XCB} \\
[3] \textbf{CHKC+C} \\
[4] \textbf{CHKSNAK+SNAPSHOT} \\
[5] \textbf{CHKCYLE+CYCLE} \\
\textbf{V}

\textbf{VRELOAD[]}V
\textbf{RELOAD}
[1] \textit{XCB+CHXXCB} \\
[2] \textit{XB+CHXXB} \\
[3] \textbf{C+C} \\
[4] \textbf{CONVERT} \\
[5] \textbf{SNAPSHOT+CHKSNAK} \\
[6] \textbf{CYCLE+CHKCYLE} \\
\textbf{V}
\[ \text{STATE[[]]} \]
\[ \text{STATE} \]
\[ [1] \quad B \]
\[ [2] \quad C \]
\[ [3] \quad CB \]
\[ [4] \quad ' ' \]
\]
\[ \text{SNAP[[]]} \]
\[ \text{X SNAP Y} \]
\[ [1] \quad \text{CYCLE=CYCLE+1} \]
\[ [2] \quad \text{APPLY} \]
\[ [3] \quad \text{+END\times 10\#Y/X} \]
\[ [4] \quad 'SNAPSHOT=';SNAPSHOT+SNAPSHOT+1;' \]
\[ [5] \quad \text{STATE} \]
\[ [6] \quad \text{END:SOX=1} \]
\]
\[ \text{CELLS[[]]} \]
\[ \text{CELLS P} \]
\[ [1] \quad Q+(R+P)/N \]
\[ [2] \quad N+(~R)/N \]
\]
\[ \text{BUS[[]]} \]
\[ \text{BUS P} \]
\[ [1] \quad Q+(R+P)/A \]
\[ [2] \quad A+(~R)/A \]
\]
\[ \text{WITH[[]]} \]
\[ \text{Z=WITH P} \]
\[ [1] \quad Z=\wedge C[;N]=\exists((\rho N),4)\rho P \]
\]
\[ \text{CELLHAS[[]]} \]
\[ \text{Z=CELLHAS P} \]
\[ [1] \quad Z=\wedge fC[;Q]=\exists((\rho Q),4)\rho P \]
∀STATUS[·]∀
∀STATUS P
[1] C[1 2 4 ;Q]∋Φ((pQ),3)pP

∀VIDLE[·]∀
∀VIDLE
[1] STATUS ' +0 '

∀PROP[·]∀
∀PROP P
[1] C[1 2 4 ;1+Q]∋Φ((pQ),3)pP

∀TOBUS[·]∀
∀TOBUS P

∀SUCHTHAT[·]∀
∀Z+SUCHTHAT P
[1] Z¬P

∀GARBAGECOLLECTOR[·]∀
∀GARBAGECOLLECTOR
[1] ISOLATOR
[2] COMPACTOR
\begin{verbatim}
\textbf{VISOLATOR[]}\textbf{V}
\textbf{ISOLATOR}
[1] \text{GC}  \leftarrow  1 \quad \phi \quad 0 \quad 0 \quad 3 \quad 0 \quad 0 \quad 1 \quad 2 \quad 0 \quad 2 \quad 2 \quad 2 \quad [1+GC+((2,2)\phi(4\times[C[3;])\epsilon'::\Box')\epsilon'[::\Box'])]
[2] \text{GC}[2;1]+3
[3] \text{GC}[1;]+3[C[1;]
[4] C[3;((0=+_GC)\wedge C[3;])\epsilon'::\Box')/1S]'+\epsilon '
[5] C[3;((GC[1;]=0)\wedge (GC[2;]=3)\wedge C[3;])\epsilon'::\Box')/1S]'+\epsilon '
\textbf{V}

\textbf{COMPACTOR[]}\textbf{V}
\textbf{COMPACTOR}
[1] C[;S]+'0= '
[2] Q\leftarrow iS-1
[3] Q\leftarrow ((C[3;Q]=')\wedge C[3;1+Q]=')/Q
[5] C[;1+Q]+Q((pQ),4)p'+0\phi '
[6] GC[;1+Q]+1
[7] C[;1]+'CELL'
\textbf{V}
\end{verbatim}
VAPPLY[\[]\]
\n\n\n\n\nV APPLY
[1] X{3 2 p 1 \n-1} X{2} X{B
[2] X{B;1;}\n+0
[3] X{B+ 1 \n-1 X{B
[4] X{B1;}\n+0 0[1+X{B1;]}
[5] X{B2;}\n+0 0[1+X{B2;}((Q+(C[3;]='0')/\nS);2)]
[6] X{B2;}\n+0 0[1+X{B2;}]
[7] CONVERT
[8] GARBAGECOLLECTOR
[9] Q+((C[4;]=M')\nA[C[3;]='0')/\nS
[10] C[4;Q]+t
[12] R+((pN+\nS)p0
[13] CELLS(WITH '0;Q')\nWITH '0*Q'
[15] CELLS(WITH '0;P')\nWITH '0*P'
[17] C[4;(C[4;Q]=P')/Q]+\nC'
[18] R+((pN+(I+ANB=1')/\nS)p0
[19] CELLS SUCHTHAT C[3;N]='
[20] CELLS SUCHTHAT(C[3;N]='1')\nA[C[4;N]=Z'
[21] IDLE
[22] PROP '-1Z'
[23] CELLS SUCHTHAT(C[3;N]=':')\nA[C[4;N]=Z'
[24] IDLE
[25] PROP '+0Z'
[26] CELLS(WITH '0+Z')\nWITH '+0*VZ'
[27] IDLE
[28] PROP '+1Z'
[29] TOBUS '-1I'
[30] CELLS WITH '+0+P'
[31] IDLE
[32] PROP '+2Z'
[33] TOBUS '-2I'
[34] CELLS WITH '+1+Z'
[35] STATUS '+0P'
[36] TOBUS '-0:
[37] CELLS WITH '+0<Z'
[38] IDLE
[39] PROP '+0Z'
[40] TOBUS '-0<'
[41] CELLS WITH '+0<P'
[42] STATUS '+0Z'
[43] CELLS WITH '+2<Z'
[44] IDLE
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>STATUS '+0P'</td>
</tr>
<tr>
<td>46</td>
<td>TORUS '+2:'</td>
</tr>
<tr>
<td>47</td>
<td>CELLS WITH '+0&gt;0'</td>
</tr>
<tr>
<td>48</td>
<td>IDLE</td>
</tr>
<tr>
<td>49</td>
<td>PROP '+0Z'</td>
</tr>
<tr>
<td>50</td>
<td>TOBUS '+2u'</td>
</tr>
<tr>
<td>51</td>
<td>CELLS WITH '+0&gt;P'</td>
</tr>
<tr>
<td>52</td>
<td>STATUS '+0Z'</td>
</tr>
<tr>
<td>53</td>
<td>CELLS WITH '+2&gt;0'</td>
</tr>
<tr>
<td>54</td>
<td>STATUS '+0P'</td>
</tr>
<tr>
<td>55</td>
<td>TOBUS '+2;0'</td>
</tr>
<tr>
<td>56</td>
<td>CELLS SUCH THAT $(C[3;N]='+') \land C[4;N]=C$</td>
</tr>
<tr>
<td>57</td>
<td>IDLE</td>
</tr>
<tr>
<td>58</td>
<td>CELLS SUCH THAT $(C[3;N]='+') \land C[4;N]='+0Z'$</td>
</tr>
<tr>
<td>59</td>
<td>$Q+(C[3;Q+1]='+')/Q$</td>
</tr>
<tr>
<td>60</td>
<td>PROP '+0Z'</td>
</tr>
<tr>
<td>61</td>
<td>CELLS(WITH '+0;0') \land WITH '+0*0$'</td>
</tr>
<tr>
<td>62</td>
<td>STATUS '+0P'</td>
</tr>
<tr>
<td>63</td>
<td>CB[Q]+1*</td>
</tr>
<tr>
<td>64</td>
<td>XCB[Q]+1</td>
</tr>
<tr>
<td>65</td>
<td>TOBUS '-0*$'</td>
</tr>
<tr>
<td>66</td>
<td>CELLS(WITH '+2;0') \land WITH '+2*0$'</td>
</tr>
<tr>
<td>67</td>
<td>TOBUS '+2:'</td>
</tr>
<tr>
<td>68</td>
<td>CELLS WITH '+0*0C'</td>
</tr>
<tr>
<td>69</td>
<td>IDLE</td>
</tr>
<tr>
<td>70</td>
<td>PROP '+2Z'</td>
</tr>
<tr>
<td>71</td>
<td>TOBUS '-2L'</td>
</tr>
<tr>
<td>72</td>
<td>CELLS WITH '+0;0C'</td>
</tr>
<tr>
<td>73</td>
<td>IDLE</td>
</tr>
<tr>
<td>74</td>
<td>PROP '+0Z'</td>
</tr>
<tr>
<td>75</td>
<td>TOBUS '-0*$'</td>
</tr>
<tr>
<td>76</td>
<td>CELLS WITH '+1;0'</td>
</tr>
<tr>
<td>77</td>
<td>IDLE</td>
</tr>
<tr>
<td>78</td>
<td>TOBUS '-0u'</td>
</tr>
<tr>
<td>79</td>
<td>CELLS SUCH THAT $(C[1;N]='+') \land (C[2;N]='+0') \land C[4;N]='+0Z'$ \land \neg ((C[2;N]='+2') \land C[4;N]='+C'$)</td>
</tr>
<tr>
<td>80</td>
<td>$B[;Q]+C[13;Q]$</td>
</tr>
<tr>
<td>81</td>
<td>$X[B[;Q;1]+X[B[;Q;2]]+1+C[B[;Q]]$</td>
</tr>
<tr>
<td>82</td>
<td>$C[1;2;4;Q+1]+C[1;2;4;Q]$</td>
</tr>
<tr>
<td>83</td>
<td>IDLE</td>
</tr>
<tr>
<td>84</td>
<td>CELLS SUCH THAT $(C[1;N]='+') \land C[2;N]='+0Z'$ $\land C[4;N]='+Q'$</td>
</tr>
<tr>
<td>85</td>
<td>IDLE</td>
</tr>
<tr>
<td>86</td>
<td>TOBUS '+2',C[3;Q]</td>
</tr>
<tr>
<td>87</td>
<td>CELLS SUCH THAT $(C[1;N]='+') \land C[4;N]='+Z'$</td>
</tr>
<tr>
<td>88</td>
<td>$C[1;2;4;Q+1]+C[1;2;4;Q]$</td>
</tr>
<tr>
<td>89</td>
<td>IDLE</td>
</tr>
</tbody>
</table>
$[130] \ C[3;Q] \Rightarrow B[3;Q]$

$[131] \ 0 \times 1 = \rho, Q + R, Q$

$[132] \ C[;Q+1]+'0[]', 'MQ'[ ' P' \ C[4;Q]]$

$[133] \ C[4;Q]+'$

$[134] \ GC[;Q]+ 1 1$

\[ \forall \]
Appendix D

LISP-like Structure in a GPM Environment

A version of GPM was implemented on a Digital Equipment Corporation PDP-12 according to Strachey's algorithm[2.6]. Three additional machine code macros are provide: LENGTH, which returns a binary count of the characters of its first argument; HEAD, which returns the first character of its argument string (If the first character is a '(' the Head returns the parenthesized expression.); and TAIL, which returns all but the first character (or parenthesized string).

Several example macros are provided. Of particular interest is the last macro set which realizes the LISP primitives-- CAR, CDR, EQ, COND. This indicates how a tree-like structure may be embedded in a string environment.
[VAL, CHORUS]!
OLD MAC DONALD HAD A FARM
E-I-E-I-O

[VAL, VERSE]!
AND ON THIS FARM HE HAD SOME #1
WITH A #2 #2 HERE AND A #2 #2 THERE
HERE A #2 THERE A #2 EVERYWHERE A #2
[CHORUS]

[CHORUS] [VERSE, CHICKS, PEEP] [VERSE, DUCKS, QUACK] [CHORUS]!
OLD MAC DONALD HAD A FARM
E-I-E-I-O
AND ON THIS FARM HE HAD SOME CHICKS
WITH A PEEP PEEP HERE AND A PEEP PEEP THERE
HERE A PEEP THERE A PEEP EVERYWHERE A PEEP
OLD MAC DONALD HAD A FARM
E-I-E-I-O
AND ON THIS FARM HE HAD SOME DUCKS
WITH A QUACK QUACK HERE AND A QUACK QUACK THERE
HERE A QUACK THERE A QUACK EVERYWHERE A QUACK
OLD MAC DONALD HAD A FARM
E-I-E-I-O
OLD MAC DONALD HAD A FARM
E-I-E-I-O
[VAL, CONS]
! (#1)(#2)

[VAL, CAR]
!
[HEAD, #1]

[VAL, CDR]
!
[HEAD, [TAIL, #1]]

[VAL, EQ]
!
[#1, [DEF, #1, F] [DEF, #2, T]]

[VAL, COND]
!
[#1, [DEF, T, #2] [DEF, F, #3]]

[VAL, MEM]
!
[COND, [EQ, #2, [NIL]], <F>, <[COND, [EQ, A], [CAR, X]]>, <T>, <[MEM, [A], [CDR, [X]]]>, [DEF, A, #1] [DEF, X, #2]]

[NIL]!

[LIST]!
(A)((B)((D)((E)( ))))

[CAR, [CDR, [LIST]]]!
[DEF, ANOTHERLIST, [CONS, [CONS, W, X], [CONS, Y, Z]]] [ANOTHERLIST]!
((W)(X))((Y)(Z))

[CONS, [LIST], [ANOTHERLIST]]!
((A)((B)((D)((E)))))(((W)(X))((Y)(Z)))

[MEM, A, [LIST]]!
T

[MEM, C, [LIST]]!
F

[MEM, D, [LIST]]!
T